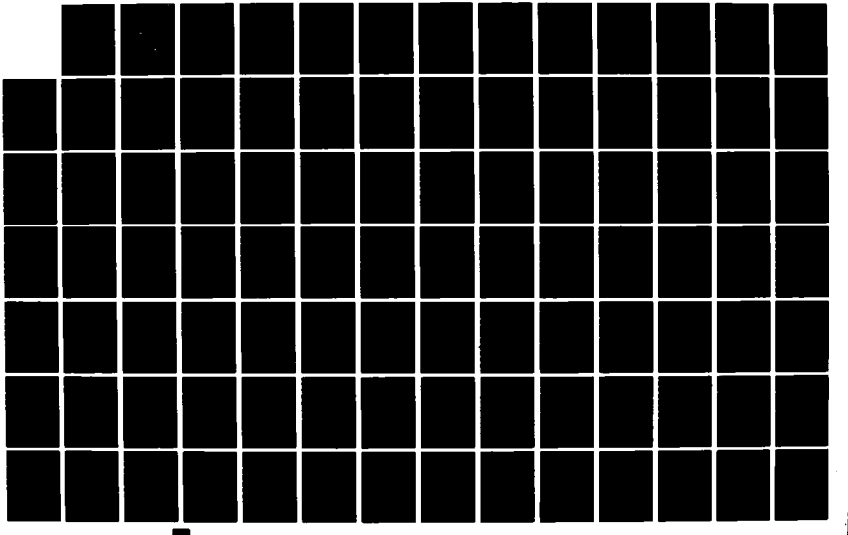
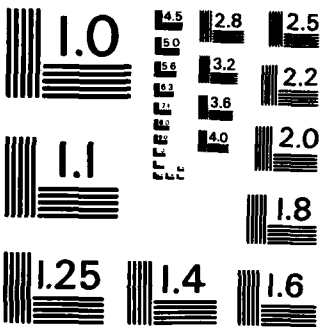
AD-A161 562 THE AREA-TIME COMPLEXITY OF SORTING(U) ILLINOIS UNIV AT 1/3
URBANA APPLIED COMPUTATION THEORY GROUP G BILARDI
DEC 84 ACT-52 N00014-84-C-0149

UNCLASSIFIED F/G 12/1 NL





REPORT ACT-52

DECEMBER 1984

THE AREA-TIME COMPLEXITY OF SORTING

GIANFRAN[illegible]

REPORT R-[illegible]

OF ILLINOIS [illegible]

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: Unclassified	1b. RESTRICTIVE MARKINGS: None
2a. SECURITY CLASSIFICATION AUTHORITY: N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release, distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): ACT-52 UILU-ENG 84-2218 R-report # R-1024	5. MONITORING ORGANIZATION REPORT NUMBER(S): N/A
6a. NAME OF PERFORMING ORGANIZATION: Coordinated Science Laboratory University of Illinois; 6b. OFFICE SYMBOL (If applicable): N/A	7a. NAME OF MONITORING ORGANIZATION: IBM Fellowship, National Science Foundation, Joint Services Electronics Program
6c. ADDRESS (City, State and ZIP Code): 1101 W. Springfield Avenue Urbana, IL 61801	7b. ADDRESS (City, State and ZIP Code): 800 N. Quincy Street Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: IBM Fellowship, NSF, JSEP; 8b. OFFICE SYMBOL (If applicable): N/A	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: N00014-84-C-0149
8c. ADDRESS (City, State and ZIP Code): 800 N. Quincy Street Arlington, VA 22217	10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT NO.
N/A	N/A	N/A	N/A

11. TITLE (Include Security Classification): The Area-Time Complexity of Sorting

12. PERSONAL AUTHOR(S): Gianfranco Bilardi

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	December 1984	

16. SUPPLEMENTARY NOTATION: N/A

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This thesis studies the minimum area $A = \alpha_{n,k}(T)$ required by a layout of a VLSI circuit that sorts n k-bit keys in time T.

The square tessellation technique is introduced as a powerful tool to establish area-time lower bounds, based on the information exchanged across the boundary of a suitable set of square cells that tessellate the layout region. When the information exchange is due to the fact that variables output on one side of the cell boundary are functions of variables input on the other side, the square tessellation yields bounds on the AT^2 measure. When, on the other hand, the information exchange is due to the fact that the cell saturates its storage resources and sends some information outside for temporary storage, the square tessellation yields bounds on the AT measure. Both AT^2 and AT lower bounds are obtained for sorting. The former dominate in fast computations, while the latter dominate in slow computations.

The analysis indicates that the nature of the problem varies considerably with the

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS ☐	

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

SECURITY CLASSIFICATION OF THIS PAGE

relative size of n and k, and suggests a classification of keys into short ($k \leq \log n$), long ($k \geq 2 \log n$), and of medium length.

Optimal or near-optimal designs of VLSI sorters are proposed for the entire range of n, k, and T, confirming the inherent validity of the lower-bound analysis.

THE AREA-TIME COMPLEXITY OF SORTING

Gianfranco Bilardi, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1985

This thesis studies the minimum area $A = \alpha_{n,k}(T)$ required by a layout of a VLSI circuit that sorts n k-bit keys in time T.

The square tessellation technique is introduced as a powerful tool to establish area-time lower bounds, based on the information exchanged across the boundary of a suitable set of square cells that tessellate the layout region. When the information exchange is due to the fact that variables output on one side of the cell boundary are functions of variables input on the other side, the square tessellation yields bounds on the AT^2 measure. When, on the other hand, the information exchange is due to the fact that the cell saturates its storage resources and sends some information outside for temporary storage, the square tessellation yields bounds on the AT measure. Both AT^2 and AT lower bounds are obtained for sorting. The former dominate in fast computations, while the latter dominate in slow computations.

The analysis indicates that the nature of the problem varies considerably with the relative size of n and k, and suggests a classification of keys into short ($k \leqslant \log n$), long ($k \geqslant 2\log n$), and of medium length.

Optimal or near-optimal designs of VLSI sorters are proposed for the entire range of n, k, and T, confirming the inherent validity of the lower-bound analysis.

ACKNOWLEDGEMENTS

My advisor, Professor Franco P. Preparata, has had a profound influence on this thesis and on my scientific development.

Many discussions with Scot Hornick, Xiaolong Jin, Majid Sarrafzadeh, Alberto Segre, Prasoon Tiwari, and Ioannis Tollis are reflected in this work in various ways. To Alberto, Majid and Scot, I am also particularly indebted for their help in preparing the final version of this manuscript.

Several conversations with Tom Leighton have provided me with considerable insight on VLSI sorting; it has been very beneficial to learn of Tom's results in their early stages.

Phyllis Young and Mary Runnells have been very patient in typing this manuscript.

The work reported in this thesis has been mainly supported by an IBM fellowship, with partial support, at different times, by National Science Foundation grant MCS 81-05552 and by the Joint Services Electronics Program under contract N00014-84-C-0149.

Finally, I want to express my gratitude to Marinella, my wife, for her love and support.

TABLE OF CONTENTS

THE AREA-TIME COMPLEXITY OF SORTING

BY

GIANFRANCO BILARDI

Laur., Università di Padova, 1978
M.S., University of Illinois, 1982

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1985

Urbana, Illinois

CHAPTER 1

INTRODUCTION

1.1 VLSI COMPUTATION

The breakthroughs in the field of electronic devices, which have lead to *Very-Large-Scale-Integration* (VLSI) technology, open new avenues to the system designer in almost all areas of electrical engineering [MC79, Mu82]. New system-theoretic concepts are necessary to take full advantage of the new technological potential, although existing theories will be an invaluable starting point, according to a pattern typical in scientific research.

We shall focus on computing systems, which will be among the first to be affected by the VLSI technological revolution. However, considering that communication and control systems make an increasing use of digital techniques for signal processing (a particular kind of computation), we realize that computing is fundamental for all electrical engineering.

The main feature that makes VLSI a very attractive environment for computing systems is the possibility to deploy - at a reasonable cost - a large number of processors cooperating in the execution of a given task. This possibility has been long pursued in the hope of increasing the system's computational throughput by means of concurrency of operations.

A sytematic development of the notion of concurrency implies a radical departure from the architecture of the traditional Von Neumann computer, and from the sequential nature of the corresponding algorithms, given as sequences of very elementary instructions each of which is to be executed in succession by the same processor. The departure from the uni-processor architecture poses the fundamental question of how to interconnect many processors so that they can efficiently exchange information when cooperating in solving a given problem. The interconnection network is in fact the most relevant feature of a parallel architecture and strongly constrains its computational capabilities. A formal way

to view a parallel architecture consists of associating to it a graph whose vertices correspond to processors and whose arcs correspond to data path. The attempt to define a general purpose architecture, whose interconnection network can support any processor-to-processor data transfer that an algorithm may require, leads to consider the fully interconnected graph. This architecture, under an equivalent formulation known as Shared Memory Machine, has in fact been extensively used in the first theoretical studies of parallel computing. However, practical considerations on limited fan-in and fan-out factors, and also on cost-effectiveness, show that the fully interconnected architecture is not a realistic model for a computer, and motivate the investigation of architectures with simpler interconnection that still support efficiently the execution of parallel algorithms. Thus we are led to the following situation. Each computational problem calls for the joint design of an algorithm and an architecture. The "best" architecture may change with the problem. Only *a posteriori* after careful analysis of many problems, may we find out whether there are general-purpose, or at least broad-purpose architectures, which are efficient for a large class of problems.

In this context an appraisal of a design must be based not only on algorithmic performance, typically characterized by time complexity, but also on some other measure capturing the "architectural complexity". The traditional count of processors is not an adequate measure because it totally disregards the communication aspects of the system. Other mathematically reasonable candidates could be related to the number of edges, or to the maximum degree, or to the diameter of the interconnection graph. However, none of these measures seems to reflect completely the cost of actually building the architecture in any technology of current interest.

It is then of the greatest theoretical interest the fact that VLSI technology naturally offers an attractive measure of architectural complexity, the *chip area*. Due to the integrated nature of VLSI technology, where processing elements (transistors) and communication elements (wires) are realized in the same medium (the silicon chip), chip area effectively accounts for the cost of all relevant aspects of the system, and its minimization is a major concern in industrial applications.

The fact that the architecture to solve a problem is not given in advance - as it was in traditional algorithm design for the Von Neumann machine - brings a new interesting consequence: the architectural complexity can be traded for the efficiency of the computation. In the context of VLSI computation, this phenomenon takes the form of *area-time trade-off*, and plays a central role in the theory.

The rigorous development of a computation theory for VLSI rests on the definition of a *model of computation* that captures the essential traits of the technology and allows for mathematical treatment of system design. A VLSI model of computation is today available, as the result of the effort of several authors [T80,BK81,Se73,Sa79,CM81,BPP82,BL84], and will be described in the next section. The assumptions of the model will be stated axiomatically. For their justification, which is sometimes based on a rather delicate and subtle analysis, we refer the reader to the literature cited above.

1.1.1 The VLSI Model of Computation

A VLSI chip can be viewed as a computation graph whose vertices are information processing devices and whose arcs are wires, that is, electrical connections responsible for information transfer as well as for power supply and distribution of timing waveforms. A given computation graph is to be laid out in conformity with the rules dictated by technology. The essence of these rules is formally accounted for in the model as follows.

Area Assumptions

(A1) (*Wire Area*): All wires have minimum width $\lambda > 0$ (which includes both the actual wire width and the clearance between wire and any other chip region), and at most ν wires (ν an integer ≥ 2) can overlap at any point (hypothesis of bounded number of layers).

(A2) (*Transistor-Port Area*): Transistors and I/O ports have minimum areas $c_T \lambda^2$ and $c_P \lambda^2$, respectively, for constants c_T and c_P.

(A3) (*Chip Area*): The chip area is at least the sum of the area of the wires, of the transistors, and of the I/O ports, and it is at most the area of the smallest rectangle (or convex region) enclosing a legal layout of the graph.

The area assumptions allow a straightforward appraisal of the area of any given design. To appraise the computation time of an algorithm we need some assumption on the timing of elementary actions, as the gate switching and the signal transmission on wires. For simplicity, in the sequel switching time is subsumed under propagation time.

Time Assumptions

(T1) (*Propagation Time Along a Wire*): A bit requires a constant time τ to propagate along a wire, irrespective of its length (*synchronous model*).

(T2) (*Algorithm Time*): The computation time of an algorithm is the time of the longest sequence of wire propagation times between beginning and completion of the computation.

Assumption (T1) is not immediate to jutify, and it is in fact false at the physical level. The essence of the justification is that, although a detailed analysis of the electric phenomenon of wire propagation [BPP82] shows that constant transmission time can be achieved on long wires only if proportionately large driving transistors are deployed, results of layout theory [BL84] ensure that a layout of the computation graph can always be found in which large drivers can be accommodated without substantial degradation of area and time performance.

The transfer of information within the chip is constrained not only by wire bandwidth, but also by fan-in and fan-out capabilities of logic gates. This fact is accounted for by the following assumptions.

Fan-in and Fan-out Assumptions

(F1) (*Bounded Fan-in*): The number of input lines of a logic gate is upper bounded by a constant f_i.

(F2) (*Bounded Fan-out*): The number of output lines of a logic gate is lower bounded by a constant f_o.

Other assumptions are often stated in the VLSI computation literature when studying lower and upper bounds for specific problems. These assumptions are not dictated by technological constraints, but rather by reasons of various kinds, for instance to avoid trivial or meaningless solutions, to enforce

features that are appealing for practical application, or to simplify the analysis. Most of these auxiliary assumptions concern the I/O protocol. We list here the most common ones:

Protocol Assumptions

(P1) (*Semellective Protocol*): The input data of the problem are available only once at the input ports.

(P2) (*Time-Determinate Protocol*): Input and output data are available at prespecified (instance independent) time.

(P3) (*Place-Determinate Protocol*): Input and output data are available at prespecified (instance independent) ports.

(P4) (*Boundary Protocol*): All I/O ports are on the boundary of the layout region.

(P5) (*Word-Local Protocol*): All the bits of a given input word enter the chip at the same input port.

Unless explicitly stated otherwise, assumptions on area, time, fan-in and fan-out, assumptions P1, P2, and P3 on I/O protocols will hold throughout this thesis . Instead, P4 and P5 will always be explicitly mentioned when adopted.

It is worth observing that, although all our networks will exhibit bounded fan-in and bounded fan-out, assumptions F1 and F2 will not be needed in most of our lower-bound proofs.

Usually, when discussing asymptotic analysis, the specific values of some of the constants in the model, such as c_T, c_P, λ, and τ, are not relevant, and can all be conventionally chosen equal to one.

It is also convenient, when considering layouts of computation graphs, to restrict the attention to embeddings on a suitable rectangular grid. Generally, this restriction could be easily removed at the price of more elaborate proofs, which would not add particular insight to the analysis.

1.1.2 The VLSI Complexity of a Computational Problem

Once a model of computation for VLSI is defined, algorithms for various problems can be proposed and analyzed, and a coherent theory can be developed. Several authors have proposed performance measures, typically a function of the area A and of the time T of the form AT^{α}, with respect to which

optimality can be defined. In our opinion, however, the following approach is more fundamental.

Given a computation problem Π, to any chip that solves Π for an input of size n, in time T_0, and area A_0, we can associate a point of coordinates (T_0, A_0) in a plane, which we call *time-area plane*. The set of all designs corresponds then to a region in this plane. The objective of VLSI complexity theory is then the determination of such region. Since if a point (T_0, A_0) is feasible then (T_0, A) is feasible for any $A > A_0$ (just waste some area!), the objective can be reformulated as follows.

Given a computation problem Π, its VLSI complexity is described by the family of curves $A = \alpha_n(T)$, one for each value n of the problem size, where $\alpha_n(T) \triangleq \min\{A_0$: there is a chip that solves Π on instances of size n with performance $(T, A_0)\}$.

Usually there is a minimum value $T_{min}(n)$ of the computation time below which no feasible design exists, and a maximum value $T_{max}(n)$ above which $\alpha_n(T)$ is constant, meaning that no savings in area result from slowing down the computation. In conclusion we would like to find, for a given problem, the value of $\alpha_n(T)$, for $T \in [T_{min}(n), T_{max}(n)]$. Typically $\alpha_n(T)$ is determined within a constant factor by establishing suitable lower and upper bounds. As expected, $\alpha_n(T)$ is increasing in n and decreasing in T, expressing the fact that a faster computation requires more computing resources.

1.2 PROBLEM STATEMENT AND ORGANIZATION OF THE THESIS

1.2.1 Sorting

Sorting is a fundamental combinatorial operation, and is among the most frequently performed by computing systems. Thus, the VLSI complexity of sorting has received a lot of attention by researchers. But, in spite of intensive study, this problem does not cease to offer extremely intriguing questions, and to reveal heretofore unsuspected facets.

Formally, the *(n,k)-sorting* problem is defined as follows:

(1) The input is a sequence of n k-bit keys, each a member of a finite set of integers.

(2) The output is a rearrangement of the input keys, so that they form a nondecreasing sequence.

Throughout this thesis, we represent the input of the (n,k)-sorting problem as an $n \times k$ array of binary variables

$$X = \{X_{i,j} \mid i = 0, 1, \ldots, n-1;\ j = 0, 1, \ldots, k-1\}$$

where $X_{i,j}$ is the coefficient of 2^j in the binary representation of the i-th input key. The i-th row of X, denoted by X_i, represents the i-th input key, and the j-th column of X, denoted by X^j, represents the j-th least significant position. A similar notation is adopted for the output array Y.

One could be tempted to analyze the complexity of sorting as a function of nk, the total number of input bits. However, as will be fully substantiated in the following chapters, the nature of sorting is strongly influenced by the relative size of n and k. Thus, it is appropriate to state the objective of our study as the determination of the minimum area $A = \alpha_{n,k}(T)$ sufficient to lay out a circuit that solves the (n,k)-sorting problem, as a function of n and k.

1.2.2. Thesis Outline

This thesis is organized in two parts, respectively devoted to the study of lower and upper bounds to the area-time complexity of sorting.

In Chapter 2, after a review of known area-time lower-bound techniques, we study the subject of multiset encoding, which turns out to be deeply related to sorting. In fact, although the input to the (n,k)-sorting problem is given as a sequence of keys, the output depends exclusively on the multiset underlying the input sequence. In the VLSI environment, where computation is governed by the flow of information in the two-dimensional chip, the information-theoretic content of the input multiset has a fundamental influence on the area-time complexity of sorting. The fact that this information content is very sensitive to the relative sizes of n and k is the primary reason for which the nature of sorting is strongly dependent on the length of the keys.

The traditional bisection flow technique is not adequate to study the area-time complexity of sorting, except for a special interval of key lengths. In Chapter 3 we introduce the notion of square

tessellation, a partition of the layout region into square cells of identical size, and we show how to obtain area-time lower bounds in terms of the information exchanged across the boundary of the tessellation cells. A novel feature of these bounds is that their form depends upon the nature of the mechanism forcing the information exchange. When the information exchange is due to the fact that the variables output on one side of the cell boundary are functions of variables input on the other side, the square tessellation technique yields lower bounds on the AT^2 measure. This mechanism has been extensively studied in the literature, especially in connection with the bisection technique. In addition to it, we consider here for the first time another mechanism, which we call *saturation*, occurring when a cell of the tessellation fills all its storage in the course of the computation, and sends some information to the rest of the chip for the only purpose of temporary storage, to request it back at a later time. When the information exchange is due to saturation, the square tessellation technique yields bounds on the AT measure.

The effectiveness of the general techniques developed in Chapter 3 is demonstrated in Chapter 4, where several lower bounds are obtained for two problems: cyclic shift and sorting. Here the keys are classified into short ($k \leqslant \log n$), long ($k \geqslant 2\log n$), and medium-length. Medium-length keys have been heretofore the object of investigation, and can be adequately studied by bisection techniques. It is for short and long keys that the full power of the square tessellation techniques becomes evident. For both cases, AT^2 and AT lower bounds can be established, and it is interesting to observe that the AT^2 bound dominates in fast computation, while the AT bound dominates in slow computation. In the last section of Chapter 4 we obtain bounds for the problem of comparison exchange, a special case of sorting where the keys are just two. The bound is on the $AT/\log A$ measure, and rests crucially on the bounded fan-in assumption, unlike the bounds mentioned above that hold even for circuits with unbounded fan-in and fan-out.

In Chapter 5 we turn our attention to upper bounds, and review some well known parallel algorithms for sorting, as well as some networks of processors particularly suited to VLSI implementations.

In Chapter 6 we study (n,k)-sorting for $k = \log n + \theta(\log n)$. After explaining why this particular value of keylength plays a central role in the construction of sorting circuits, we turn our attention to specific designs. We first consider the bitonic sorting algorithm, and propose two architectures, the pleated cube-connected-cycles, and the mesh of cube-connected-cycles, both of which achieve optimal area-time performance in a wide spectrum of computation times. The fastest bitonic sorter works in time $T = \theta(\log^2 n)$. To obtain faster sorters we then turn our attention to another algorithm, the merge-enumeration combination. A network that combines the cube-connected-cycles and the orthogonal-trees architectures executes this algorithm in $\theta(\log n)$ time and optimal area.

In Chapter 7 we consider the (n,k)-sorting problem for arbitrary k, and we propose three sorting networks, respectively tailored to short, medium-length, and long keys. The algorithms presented in this chapter are new. The ones for short and medium-length keys exploit efficient encodings of schemes for multisets, while the algorithm for long keys takes advantage of the non-word-locality of the I/O protocol. The fact that the resulting VLSI designs are optimal or near-optimal confirms the inherent validity of the lower-bound analysis developed in Chapters 3 and 4.

Some closing remarks are finally presented in Chapter 8.

PART I

LOWER BOUNDS

CHAPTER 2

PRELIMINARIES

2.1 INTRODUCTION

Part I is devoted to the study of lower bounds on the area-time complexity of sorting. However, the techniques that we develop are general, and will probably be useful to investigate several other problems.

We recall from Chapter 1 that the VLSI complexity of a computational problem Π is described by the family of functions

$$A = \alpha_n(T), T \in [T_{min}(n), T_{max}(n)] \qquad 2.1$$

where n is the input size, $\alpha_n(T)$ is the area of the smallest design that solves Π in time T, T_{min} is the minimum time required to solve Π (regardless of the area), and T_{max} is a time such that, for $T > T_{max}$, $\alpha_n(T)$ is constant with respect to T.

Area-time lower bounds can be stated in different forms. The most common are

$$A = \Omega(f_1(n, T)). \qquad 2.2$$

$$T = \Omega(f_2(n, A)). \qquad 2.3$$

$$g(A, T) = \Omega(f(n)). \qquad 2.4$$

where f_1, f_2, g, and f are suitable functions. It is usually a simple matter to convert one of the above forms into another. The choice of the form to be used in a specific case is only a matter of convenience.

2.1.1 Layout Theory

Since a VLSI chip can be viewed as the layout of a given computation graph, some useful tools to establish area-time lower bounds can be borrowed from layout theory, a chapter of graph theory which studies, among other things, the problem of determining the minimum area needed to embed a given graph in the plane, according to some specified layout rules.

Typically, lower bounds (and also upper bounds) on the layout area are given in terms of some auxiliary quantities associated with the graph, which are hopefully easier to compute or to bound than the area itself. Among the most interesting auxiliary quantities proposed in the literature are the bisection width [T80], the crossing number [L81a], the wire area [L81a], the separator [Ls80b, Va81], and the bifurcator [L82,BL84].

When applying layout theory to obtain area-time lower bounds, we do not deal with a specific graph, but with all the graphs that can support the computation to solve a given problem Π, in a given time T. Thus, our goal is to show how this computational property of the graph implies a bound, either directly on the area, or on some related auxiliary quantities. Some techniques have been proposed in the literature to achieve this goal, and we briefly review them in the next section.

2.1.2 Area-Time Lower-Bound Techniques

To date, all known area-time lower bounds belong to one of the three following classes.

(1) *Input-output bounds.* They are of the form

$$AT = \Omega(size\ of\ input + size\ of\ output) \qquad 2.5$$

and are a trivial consequence of the fact that the area is at least proportional to the number of I/O ports, which in turn is at least proportional to the maximum number of bits that the chip inputs or outputs in a time unit. For boundary chips, (where all the I/O ports are placed on the boundary of the layout region), the I/O bound becomes

$$pT = \Omega(size\ of\ input + size\ of\ output) \qquad 2.6$$

where p is the perimeter of the layout region. Bound (2.6) is usually combined with other considerations to obtain area-time bounds.

(2) *Functional dependence bounds.* Functional dependence of the output variables on the input variables can sometimes be exploited to strengthen the I/O bound, as in [Jh80], where it has been shown that, for the addition of binary integers with n bits,

$$A = \Omega(\frac{n}{T}\log(\frac{n}{T})) \qquad 2.7$$

or equivalently,

$$AT/\log A = \Omega(n). \qquad 2.8$$

The argument to establish the bound is rather subtle, and we will discuss it in detail in Section 4.3, where we apply it to the problem of comparison exchange.

(3) *Information-Exchange Bounds.* Almost all the nontrivial known lower bounds on area-time complexity are of the type

$$AT^2 = \Omega(I^2(n)). \qquad 2.9$$

where $I(n)$ is the bisection-information of the problem Π being considered, a very important notion introduced by Thompson [T80]. Informally, the bisection width b of a graph G = (V,E), is the minimum number of edges to be removed in order to separate a set of $|V|/2$ vertices from its complement. (For formal definitions and generalizations see [T80], and also Section 3.1.) The bisection-information arguments are based on two facts: (i) the layout area is at least proportional to the square of the bisection width; (ii) any computation graph that solves a given problem Π must support an information exchange $I(n)$ through its bisection, where $I(n)$ is a function associated with Π. The bound (2.9) follows easily from (i) and (ii), considering that $b \geq I(n)/T$. The evaluation of $I(n)$ requires an argument tailored to the particular problem being studied. Indeed, considerable attention has been devoted to the subject of information exchange, which we survey briefly in the next section.

2.1.3 Information exchange

In recent years the study of both distributed and VLSI computing has generated considerable interest in the analysis of the amount of information that different processors have to exchange when cooperating in solving a given problem.

Several quantitative definitions of the information exchange I associated to a problem Π have been proposed, and several techniques to lower bound I have been developed. The objective of this section is to recall the main concepts at an intuitive level, and to indicate the appropriate references, where a more detailed treatment of the subject can be found.

The general framework is one in which two processors P_1 and P_2 cooperate to solve a problem Π, or equivalently to compute a function f. The basic question is: How many bits do the processors have to exchange during the computation? The answer is obviously dependent on a number of assumptions, and different authors have made different assumptions. We list some of them here.

I/O-Variable assignment. In the simplest case the assignment of I/O variables to processors is completely specified. In applications to VLSI we are typically interested in a class of assignments, and the information exchange must be minimized over the class. ([Y79], [T80], [BK81], [AA80], [Y81], [LS81], [BG82], [MS82], [Sa79], [Vu83], [JK84], [AUY83].)

Communication protocol. We may consider a one directional link, say from P_1 to P_2 (one-way communication) or a link for each direction (two-way communication), ([Y79]). We may also impose bounds on how many messages can be exchanged, a message being a run of bits sent by P_i to P_{3-i}. Alternatively, we may bound the length of the messages, and so forth ([PS82], [DGS84]).

Type of computation. The computation performed by P_1 and P_2 can be assumed to be deterministic, or nondeterministic, or randomized (Las Vegas). ([MS82], [PS82], [LS81], [DGS84], [Y75], [AUY83].)

Complexity measure. Finally, we can count the bits exchanged by P_1 and P_2 in the worst case instance, or in several kinds of average case [Km83].

The above list of assumptions is by no means exhaustive, but should give an idea of the variety of issues which are addressed in this area of study. Typical results that can be found in the literature concern: (i) general lower-bound techniques; (ii) bounds on the information exchange of specific functions; (iii) the study of complexity classes related to various definitions of I; and (iv) conditions under which the bound $AT^2 = \Omega(I^2)$ is valid in the VLSI model.

A complete account of the theory of information exchange is not our present objective. However, we will return to this subject to propose some new developments, with relevant applications to VLSI complexity.

2.1.4 Summary of Part I

The input to a sorting problem is a multiset, and for this reason efficient schemes to encode multisets are essential to obtain good algorithms. Moreover, the fact that the efficiency of a given encoding scheme is very sensitive to the ratio between the size of the multiset and the size of the universe from which the elements are drawn, makes the nature of the sorting problem vary considerably with the length of the keys being sorted. Thus, both lower-bound arguments and upper-bound constructions greatly benefit from a solid understanding of the subject "encoding of multisets" which is treated in Section 2.2.

Chapter 3 is devoted to general lower-bound techniques. In Section 3.1 we generalize the notion of bisection width by introducing the notion of dichotomy width of a graph, a quantity very useful to lower bound the layout area of some graphs. In Section 3.2 we show that a suitable generalization of the traditional concept of information exchange can be used to lower bound the dichotomy width of computation graphs. When combined with those of Section 3.1, these results provide powerful tools to lower bound the area-time complexity of computational problems.

The traditional bisection-information techniques as well as the generalization proposed in Section 3.2 capture the idea that if some variables output at a given place carry information on other vaiables input at a different place, then some kind of information flow between the two places will be required

by the computation. However, there are cases where the information is input in a place close to where it must be output, and nevertheless it must be temporarily transferred to a different place, due to the fact that all local storage is saturated. In Section 3.3 we show how this intuition can be formalized by defining the notion of information exchange under bounded storage. We also develop a general technique to obtain area-time lower bounds based on this notion.

In Chapter 4 we apply the results of Chapter 3 to specific problems. In Section 4.1 we derive lower bounds for the information exchange and the area-time complexity of cyclic shift. Although cyclic shift is an interesting problem in its own rights, our main motivation to analyze it is due to the relationship between cyclic shift and sorting, to be systematically exploited in Section 4.2 where we finally concentrate on the sorting problem.

Section 4.2 is organized in three subsections, respectively devoted to the study of three different ranges of key lengths. Several new lower bounds are obtained both on the AT^2 measure (using the dichotomy-information technique), and on the AT measure (using the saturation technique). As we shall see, the AT^2 bounds dominate in fast computations, whereas the AT bounds dominate in slow computations.

Finally, in Section 4.3 we discuss the area-time lower bounds for the comparator-exchanger, which can be viewed as a sorter of two keys. Here we have to investigate the notion of functional dependence and its effect on the area-time performance. Crucial to this type of argument is the notion of bounded fan-in digital circuits.

2.2 ENCODING MULTISETS

This section is devoted to the study of efficient encodings of multisets. We are interested in multisets because:

(i) The input to a sorting problem is a multiset (the ordering of the elements in the input list is immaterial).

(ii) A sorted list can be viewed as a canonical representation of the underlying multiset (two lists of elements represent the same multiset if and only if they are identical when sorted).

The study of efficient encodings of multisets will provide us with a background both for the information-based lower bounds of Chapter 4, and and for the upper bound constructions of Chapters 6 and 7.

A *multiset* S is a collection of elements from a totally ordered set U called the *universe*, with repetitions allowed. In the sequel we are only concerned with finite multisets and finite universes so that, without loss of generality, we can use the following notation:

$$S = \{X_0, X_1, \ldots, X_{n-1}\} \tag{2.10}$$

$$U = \{0, 1, \ldots, r-1\}. \tag{2.11}$$

Thus, n is the size of the multiset, and r is the size of the universe. Usually we think of the elements of U as encoded in binary, and we denote by $k = \lceil \log r \rceil$ the number of bits needed to encode an element. Since the order of the element of S is immaterial, representation (2.10) is not unique, and given any permutation $\pi(0), \pi(1), \ldots, \pi(n-1)$ of the integers $0,1,\ldots,n-1$, we can also write

$$S = \{X_{\pi(0)}, X_{\pi(1)}, \ldots, X_{\pi(n-1)}\}.$$

This representation becomes unique if we add the constraint that $X_{\pi(i)} \leq X_{\pi(i+1)}$, for $i = 0,1,\ldots,n-2$, or in other words if we require that the sequence $X_{\pi(0)}, X_{\pi(1)}, \ldots, X_{\pi(n-1)}$ be sorted in nondecreasing order. From this standpoint sorting becomes the operation of computing a canonical representation for a multiset.

Other representations are clearly possible, and could be more convenient in some situations. In particular, in VLSI computation we are interested in nonredundant representations because they require less bandwidth for transmission.

2.2.1 Counting Arguments

A simple combinatorial argument shows that the number of multisets of size n in a universe of size r is $\left| \begin{matrix} n+r-1 \\ r \end{matrix} \right|$. Thus, the number of bits necessary to encode a multiset is

$$e(n,r) = \log \left| \begin{matrix} r+n-1 \\ r \end{matrix} \right| \qquad 2.12$$

If we use Stirling's approximation for the factorial, after some manipulations we can rewrite Eq. (2.12) as

$$e(n,r) = n\log(1+r/n) + r\log(1+n/r) + \textit{lower order terms.} \qquad 2.13$$

It is interesting to consider the asymptotic behavior of $e(n,r)$ when r is an increasing function of n, as in the following examples.

(1) $r/n \to 0$ $\quad e(n,r) \approx r(\log n - \log r)$

($r = r_0 = constant$, $e(n,r_0) \approx r_0 \log n$)

(2) $r = n \times constant$ $\quad e(n,r) = \theta(n)$

($r = n$, $e(n,n) \approx 2n$)

(3) $r/n \to \infty$ $\quad e(n,r) \approx n(\log r - \log n)$

($n = n_0 = constant$, $e(n_0,r) \approx n_0 \log r$)

Certainly there are encodings of multisets that use strictly $e(n,r)$ bits. However, we are interested in encodings that either arise naturally from problems, or that, although artificially introduced, preserve some intuitive meaning, and are useful in multiset manipulations.

2.2.2 List Encoding

The most natural way to describe a multiset consists in giving a list of its elements, in any order. Clearly $e_{list}(n,r) = n\log r = nk$ bits are used for this representation. Thus, the list encoding is optimal (in the order) if and only if the universe is large enough, namely if

$$k = \log n + \Omega(\log n) \qquad 2.14$$

or, equivalently, if $r > n^{(1+\alpha)}$ for some $\alpha > 0$. The list encoding becomes very inefficient for a small universe. In the extreme case of $r = 2$, $e_{list}(n,2) = n$, whereas $e(n,2) \approx \log n$. Therefore we turn out attention to another method.

2.2.3 Multiplicity Encoding

Another simple way to specify a multiset is to say how many occurrences it contains of any given element of the universe. Formally, we introduce the multiplicity function $\mu(i)(i = 0,1,\ldots,r-1)$ of multiset S defined as

$$\mu(i) \triangleq \textit{number of occurrences of element } i \textit{ in multiset } S. \qquad 2.15$$

Since $\mu(i)$ is at most n, $\mu(i)$ can be represented with $\lfloor \log(n+1) \rfloor \leq \log n + 1$ bits, and hence S can be encoded with $e_{mult}(n,r) = r(\log n + 1)$ bits. This encoding is optimal in the order when

$$k = \log n - \Omega(\log n) \qquad 2.16$$

or,equivalently, if $r < n^{(1-\alpha)}$ for some $\alpha > 0$. Slightly better results can be obtained by using a variable length encoding for $\mu(i)$. For example we can encode integer h with $2\lfloor \log(h+1) \rfloor$ bits by using the empty string for $h = 0$. The multiplicity function can then be represented by the list $\mu(0), \mu(1), \ldots, \mu(r-1)$ with the commas encoded as '01'. Thus we can use a total number of bits

$$e'_{mul}(n,r) = \sum_{i=0}^{r-1} 2\lfloor \log(\mu(i)+1) \rfloor + 2(r-1).$$

It is easy to see that, under the constraint $\mu(0) + \mu(1) + \ldots + \mu(r-1) = n$,

$$e'_{mul}(n,r) = O(r \log(n/r + 1) + r)$$

which is optimal for $r \leq n$. For $r > n$, we must resort to different techniques.

2.2.4 The Insert-and-Prune Encoding

In this section we propose a new encoding for multisets which is based on a sorting method and is not as natural as the list and the multiplicity schemes, but it is simple and elegant. Moreover it can be effectively used in some sorting algorithms.

Let us begin with a simple observation. In a sorted sequence of n elements of k bits each, the sequence of bits in the most significant position is a run of zeros followed by a run of ones. Therefore it can be completely described by specifying how many zeros there are, which only requires $\log n$ bits instead of the n bits taken in the list representation. In general, in a sorted sequence the j-th most significant position (from the left) contains at most 2^j alternating runs of zeros or ones. Thus, for $j < \log n$ not all the binary sequences of n bits are candidates to be the j-th position of a sorted sequence, and therefore less than n bits are needed to encode that position.

We could try to exploit systematically the above observations and build an efficient encoding based on the length of runs of identical bits in each bit position of the sequence, but the resulting scheme would be rather awkward and difficult to manipulate. However, the above discussion reveals an important property: the leftmost bit positions in a sorted sequence carry less information than the number of bits devoted to these positions in the list. As it turns out, if we have some extra knowledge about our sorted sequence, we may even completely reconstruct the sequence by looking only at its least significant position! This is a consequence of the following result.

Theorem 2.1. If $S = \{X_0, \ldots, X_{n-1}\}$ is a multiset drawn from the universe $U = \{0, 1, \ldots, r-1\}$, and T is the sorted list of the union of S and U, then there is a one-to-one correspondence between S and the sequence of bits in the least significant position of T.

Proof. T is the concatenation of r subsequences the i-th of which consists of, $\mu(i) + 1$ copies of element i ($i = 0, \ldots, r-1$), where $\mu(i)$ is the multiplicity function of S. The situation is illustrated in Figure 2.1. The least significant bits of T are the concatenation of r sequences, the i-th of which consists of $\mu(i) + 1$ identical bits each equal to i modulo 2. Thus, from the least significant bits we can recover

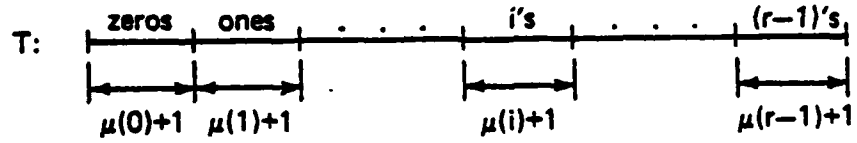


Figure 2.1. The structure of sequence T.

the multiplicity encoding of S, and S itself. The converse is obvious. □

Remark. It follows from Theorem 2.1 that the sequence of bits in the least significant position of T is a valid encoding of S, requiring $n + r$ bits. For $r = n$ the encoding is optimal up to lower order terms. For $r >> n$ or $r << n$ the encoding is highly inefficient. However, for $r > n$, the following generalization of Theorem 2.1 yields a better result.

Theorem 2.2. Let for simplicity $n = 2^v$, $r = 2^k$, and $s = 2^\sigma$ be powers of two. Let also $S = \{X_0, \ldots, X_{n-1}\}$ be a multiset from the universe $U = \{0, \ldots, r-1\}$, and $U(s) = \{0, s, 2s, \ldots, r-s\}$ be a sampling of U with period s. Define T as the sorted list of the union of S and $U(s)$. Then, there is a one-to-one correspondence between S and the sequence formed by the $\sigma + 1$ least significant bits of the elements of T.

Proof. We introduce the notation

A' = *multiset of the prefixes of length* $k - \sigma$ *of the elements in multiset* A

and we define $U(s)'$ and T' accordingly. Clearly $U(s)' = \{0, 1, \ldots, r'-1\}$ where $r' = r/s$. Thus we can apply Theorem 2.1 to multiset S' and universe $U(s)'$, to reconstruct T' from the $(k - \sigma)$-th most significant bit position of T. Then we easily reconstruct the entire T by concatenating most and

least significant positions of each element. Finally we obtain $S = T - U(s)$. □

Remark. Theorem 2.2 reduces to Theorem 2.1 when $\sigma = 0$.

We call *insert-and-prune* encoding the representation of S obtained by augmenting S with $U(s)$ and sorting the result (therefore effectively *inserting* the sorted $U(s)$ into the sorted S), and by subsequently removing (*pruning*) the $k - \sigma - 1$ most significant bits of each element. The number of bits required by this encoding is

$$e_{isp}(n,r) = (\sigma+1)(n + 2^{-\sigma}r). \qquad 2.17$$

For $r < n$, the choice $\sigma = 0$ minimizes $e_{isp}(n,r)$ giving $e_{isp}(n,r) = n + r$ and the encoding is not optimal. For $r \geq n$, the choice $\sigma = \log(r/n)$ yields an optimal encoding with

$$e_{isp}(n,r) = O(n(\log(r/n) + 1)). \qquad 2.18$$

Summary of insert-and-prune encoding. If S is a multiset of n elements of $k = \log n + h$ bits, we can encode it with $e_{isp} = (h + 1)2n$ bits by the following procedure.

1. Add to S the n elements $\{2^h i : i = 0,1,\ldots,n-1\}$.
2. Sort the resulting multiset.
3. Retain only the bits in the $(h+1)$ least significant positions.

A picture of the encoding scheme is given in Figure 2.2.

2.2.5 Two-Stage Encoding

Given a multiset S with a multiplicity function $\mu(i)$, $i = 0,\ldots,r-1$, we define the distribution function

$$M(i) = \sum_{j \leq i} \mu(i), \quad i = 0,1,\ldots,r-1. \qquad 2.19$$

Obviously $(M(0), M(1),\ldots, M(r-1))$ is a sorted sequence with all elements less than or equal to n. If $n \leq r$, we can then encode this sequence by the insert-and-prune method. Since the size of the

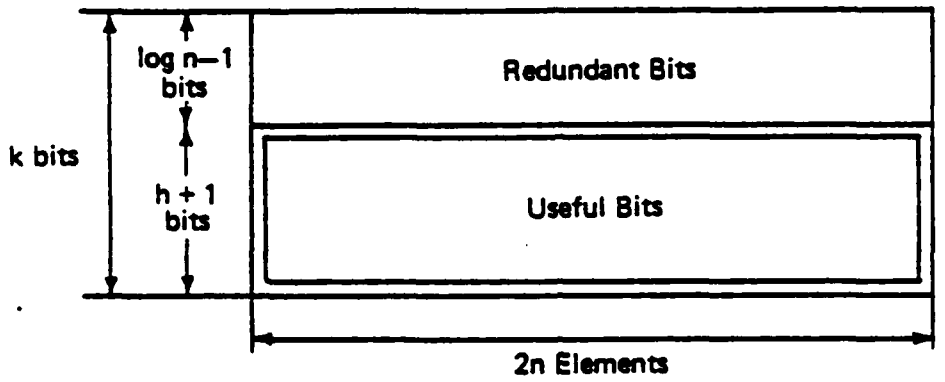


Figure 2.2. The insert-and-prune (isp) encoding scheme.

sequence is r, and the size of the elements is $\log n = \log r - 1 + (\log n - \log r + 1)$, the two-stage encoding of S uses a number of bits

$$e(n,r) = 2r(\log n - \log r + 1) = O(r \log(n/r + 1)), \qquad 2.20$$

which is optimal.

2.2.6 Summary of Optimal Encodings

We summarize the encodings described in the previous section in Figure 2.3, where we show the ranges of k in which each of the encodings is optimal. We recall that the results are of an asymptotic nature, and are based on the assumption that k increases with n.

For completeness, we report here that a multiset $S = \{X_0, \ldots, X_{n-1}\}$ can be represented by specifying the difference between consecutive elements in the sorted arrangement of S. This encoding is

efficient when $r \geq n$, and has been successfully exploited in [Lo83] to obtain optimal sorting algorithms on a distributed system.

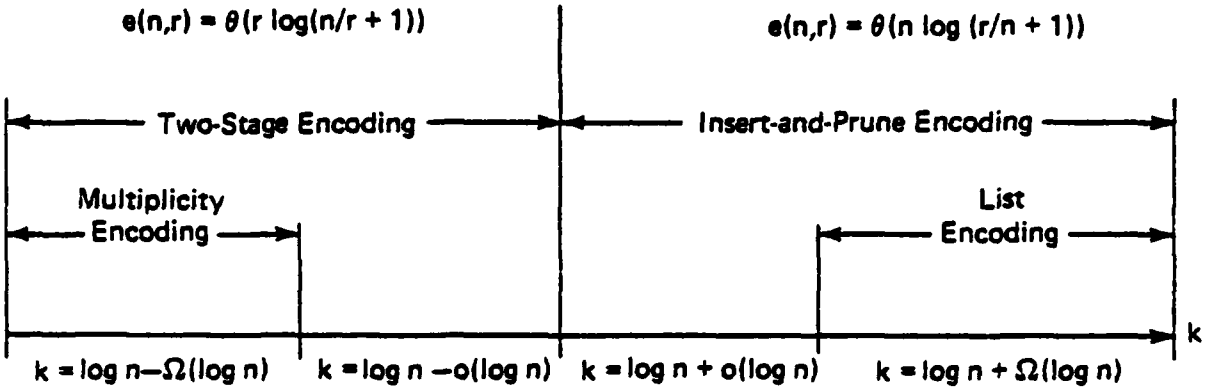


Figure 2.3. Ranges of optimality of encoding schemes for multisets.

CHAPTER 3

LOWER-BOUND TECHNIQUES

The lower-bound techniques of this chapter use a combination of a *geometric* argument, based on a suitable subdivision of the layout region, and an *information-theoretic* argument, based on the information exchange between a region of the geometic subdivision and the remaining part of the layout.

Two basic methods to subdivide the layout region will be considered.

(i) *Bipartition.* It is the classical method introduced by [T80], whereby the subdivision is obtained by cutting the layout into two regions separated by a straight line (or a simple deformation thereof).

(ii) *Square tessellation.* It is a method that we shall introduce in the next section, and consists in subdividing the layout region in a mesh of square cells all of the same size.

We shall also make use of two basic information-theoretic notions.

(a) *Information exchange.* It is the classical notion studied by several authors, as briefly reported in Section 2.1.3, and will be formally defined in Section 3.2.

(b) *Bounded-storage information-exchange.* It will be formally defined in Section 3.3 as a refinement of (a) when a bounded storage is assumed for the processors that execute the computation, and is instrumental to study information-exchange in saturation conditions.

When classified with respect to the geometric and the information-theoretic notions of which they make use, the lower-bound techniques can be of one of the four types: (i)-(a), (i)-(b), (ii)-(a), (ii)-(b).

As we shall see, types (i)-(a) and (ii)-(a) yield lower bounds on the AT^2 measure, and type (ii)-(b) yields lower bounds on the AT measure. Presently, we do not know of any useful application of technique (i)-(b).

In all the applications where we shall make of the square-tessellation technique, although we subdivide the layout into many regions, we only need to consider the information exchange occurring between one region and the rest of the layout.

Thus, in both the bipartition and the square tessellation techniques, we are effectively studying the information exchange that occurs between a set of nodes of the computation graph, and its complement. We refer to a partition of the vertex set of a graph into two sets as to a *dichotomy* of the graph. As we shall see, dichotomies, and the related notion of dichotomy-width (to be formally defined in Section 3.1) play a relevant role in lower-bound theory.

To avoid terminological confusion, we stress the point that dichotomy is a topological notion pertaining to a graph, while bipartition is a geometric notion pertaining to a layout (of a graph). The two concepts should be kept distinct, although any bipartition of the layout induces a dichotomy of the graph.

We shall use the term *bisection* only in a topological denotation, to refer to a dichotomy which is (roughly) balanced with respect to a given weight of the vertices of the graph. This is in agreement with the original definition given in [T80]. Instead, we shall *not* use the term bisection to denote a geometric cut of the layout, even if it induces a bisection in the corresponding graph.

3.1 THE DICHOTOMY LOWER BOUND ON THE LAYOUT AREA

In this section we present a new technique to obtain lower bounds on the layout area of graphs. The technique is based on the notion of dichotomy which generalizes the notion of bisection.

Given a graph $G = (V, E)$ we call *dichotomy* a partition $D = (V_1, V_2)$ of the vertex set V, and we denote by $\delta(D)$ the number of edges of G that connect V_1 to V_2. We define the *dichotomy width* with respect to a class Γ of dichotomies of G, as the minimum number of edges that have to be removed in order to disconnect V_1 from V_2 over all dichotomies in Γ. Formally we have the following definition.

Definition 3.1. Given a graph $G = (V, E)$, and a class Γ of dichotomies of G, the Γ - *dichotomy width* is

defined as

$$\delta_\Gamma \triangleq \min_{D \in \Gamma} \delta(D). \tag{3.1}$$

Remark. If $\Gamma=\{D:|V_1| = \lfloor N/2 \rfloor\}$, where $N=|V|$, then δ_Γ becomes the minimum bisection width as defined by Thompson [T80].

In the sequel we consider some choices of Γ that enable us to prove a lower bound on the layout area in terms of δ_Γ. We begin with the simple case in which Γ is the class of all dichotomies (V_1, V_2) with $V_1 = m$ $(m < N = |V|)$:

$$\Gamma_m \triangleq \{(V_1, V_2): |V_1| = m\}. \tag{3.2}$$

To simplify the notation, we write $\delta(m)$ for δ_{Γ_m}, the dichotomy width of G with respect to Γ_m.

We discuss now some concepts that are useful in relating the layout area A of a graph to its dichotomy width $\delta(m)$. A graph is to be laid out on the *layout grid*, a plane grid the vertices of which have integer coordinates in a suitable cartesian frame of reference. A layout of a graph is an assignment of nodes to vertices of the grid, and of edges to paths of grid edges, where different edges of G share only grid vertices. This restriction implies that all nodes have degree at most four, a property we shall always assume when discussing layouts of graphs.

Beside the layout grid, it is convenient to consider another grid, the *auxiliary grid*, the vertices of which are the points of semi-integer coordinates, as shown in Figure 3.1.

The area of a given layout is defined to be the area of its smallest enclosing rectangle with boundary on the auxiliary grid. The *layout area* of A of a graph G is the area of its smallest layout. A *zig-zag line* is either a straight line on the auxiliary grid, or a pattern of the kind shown in Figure 3.2. Formally, a vertical zig-zag line is a set of the form

$$\{(x_0, y): -\infty < y \leq y_0\} \cup \{(x, y_0): x_0 \leq x \leq x_0 + a\} \cup \{(x_0 + a, y): y_0 \leq y < \infty\}$$

where $a \in \{0,1\}$. A horizontal zig-zag line could be defined similarly.

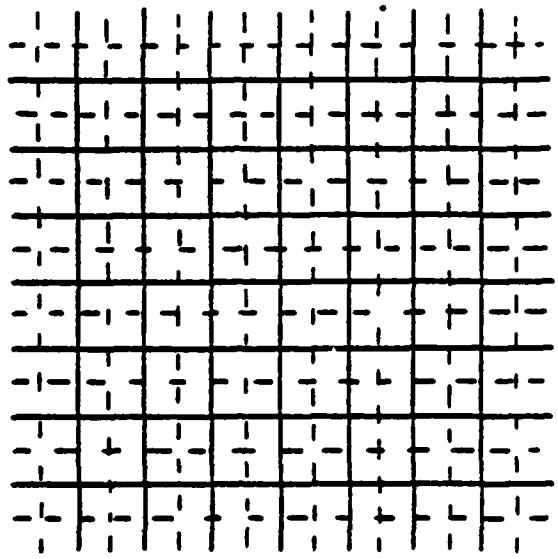


Figure 3.1. The layout grid (solid) and the auxiliary grid (dotted).

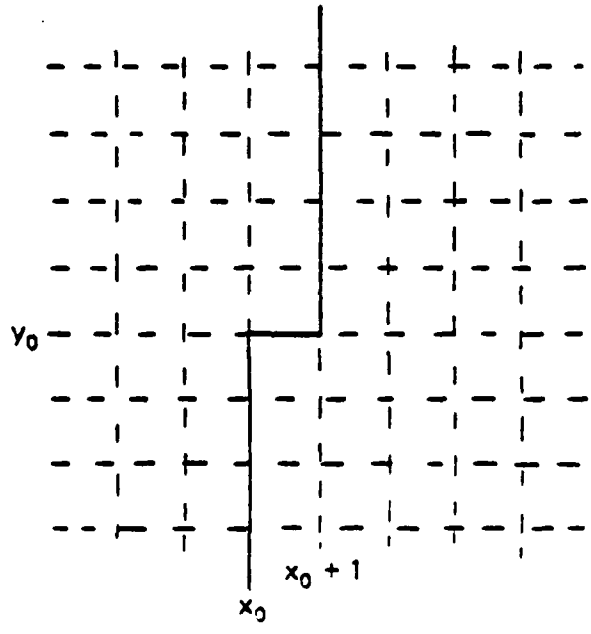


Figure 3.2. A (vertical) zig-zag line.

The next theorem states the first lower bound to A in terms of $\delta(m)$. The result generalizes the bound $A \geq (\delta(N/2) - 1)^2$ obtained by Thompson [TS0], and the proof is based on the same technique

introduced by [T80], which we call a *bipartition* technique because it is based on a suitable partition of the layout into two regions.

Theorem 3.1. If a graph G has dichotomy width $\delta(m)$, then the sides l_x and l_y of the smallest enclosing rectangle R of any layout of G have length at least $\delta(m)-1$, whence

$$A \geq (\delta(m)-1)^2 = \Omega(\delta^2(m)). \tag{3.3}$$

Proof. It is easy to show that there exists a vertical zig-zag line which splits R into two regions (separated by either l_y or l_y+1 grid segments), one containing m nodes of G, and the other $N-m$. By the definition of Γ_m-dichotomy width at least $\delta(m)$ edges cross the boundary between the two regions, and therefore $l_y + 1 \geq \delta(m)$ or $l_y \geq \delta(m)-1$. □

The next theorem provides another bound on A in terms of $\delta(m)$. The bound is better than (3.3) whenever $m = o(N)$. The proof introduces a novel technique, which we call the *square tessellation* technique, because it is based on a partition of the layout region into a mesh of square cells, all of the same size.

Theorem 3.2. For every graph $G = (V, E)$, and every $m < N$,

$$A = \Omega\left(\frac{N}{m}\,\delta^2(m)\right). \tag{3.4}$$

Proof. Given a layout of G (on the unit grid), let R be the smallest enclosing rectangle (on the auxiliary grid). Let us consider on the auxiliary grid a mesh of square cells with sides of length $l \triangleq \left\lfloor(\delta(m)-1)/4\right\rfloor$, and such that one cell has a vertex overlapping with the southwest corner of R (see Figure 3.3).

We claim that no cell of the mesh contains m or more nodes of G. In fact if a cell contains m or more nodes then we can find a zig-zag line that cuts the cells into two polygons one of which, called P, contains exactly m nodes. (See Figure 3.4.) This polygon has a perimeter $p \leq 4l = 4\left\lfloor\delta(m) - 1/4\right\rfloor \leq \delta(m) - 1$, so that less than $\delta(m)$ edges can cross it, contradicting the

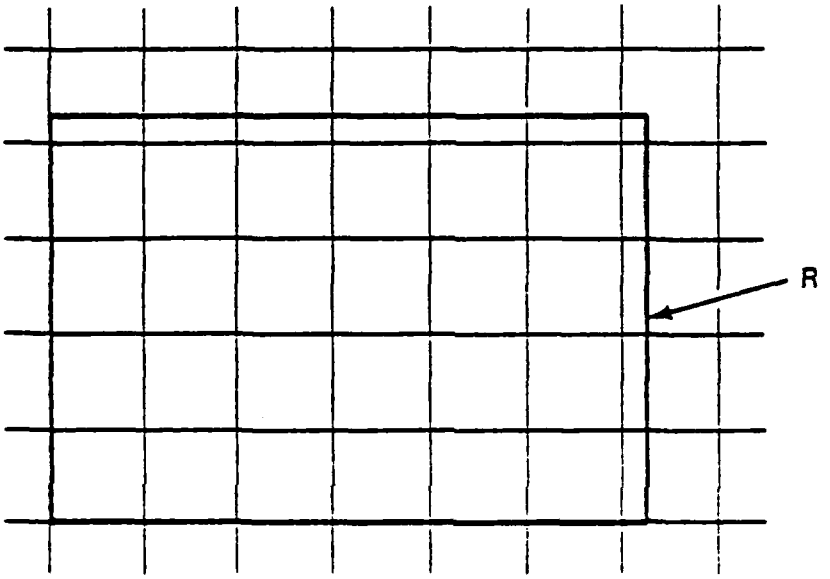


Figure 3.3. Square tessellation of the layout (to the proof of Theorem 3.2).

definition of $\delta(m)$. We conclude that at least $\lfloor N/m \rfloor$ cells of the mesh contain some nodes of G, and therefore overlap with R. The total area of these nonempty cells is then

$$A \geq \frac{N}{m} \left\lfloor \frac{\delta(m)-1}{4} \right\rfloor^2 . \qquad 3.5$$

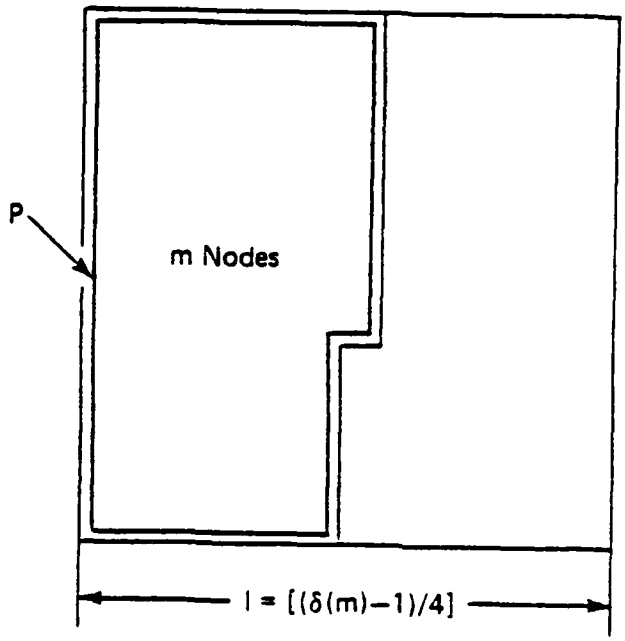


Figure 3.4. A cell with m nodes or more.

Due to some nonempty cells which have only a partial overlap with R, the layout area A can be smaller than A_c . However these cells can occur only at the boundary of R. Since Theorem 3.1 ensures that the length of each side of R is at least four times the length l of the side of the cell, R contains at least 16 cells, so that it is easy to show that $A \geq 16/25\, A_c$. Thus,

$$A \geq \frac{16}{25}\frac{N}{m}\left|(\delta(m)-1)/4\right|^2 \approx \frac{1}{25}\frac{N}{m}(\delta(m)-1)^2 \qquad 3.6$$

and the theorem is proved. □

Remark. Equation 3.6 yields a better bound than Equation 3.3 for $m < N/25$.

In general, the best bound that we can obtain for the area of a given graph G from Theorem 3.2 corresponds to the value m_0 of m that maximizes in the function $\Delta(m) \cong \delta^2(m)/m$. For most of the

computation graphs considered in the area-time literature $m_0 \approx N/2$ (or more in general $m_0 = \theta(N)$), and Theorem 3.1 is sufficient to obtain good lower bounds. This fact accounts for the success of bipartition techniques, and has lead researchers to focus almost exclusively on balanced partitions of computation graphs. However, the computation graphs that solve some important problems, including sorting, have a $\Delta(m)$ function whose maximum is achieved for values of m considerably smaller than N. In these cases, the notion of dichotomy and the square tessellation technique developed in the present section are instrumental to obtain *tight* bounds.

When applying dichotomy arguments to computation graphs, we often need to consider a class Γ more general than Γ_m . For example we may focus on the set U of the nodes that are input ports, and we may want δ_Γ to represent the minimum number of edges to be removed from G in order to disconnect a set V_1 containing m input ports from its complement V_2. In this case the appropriate definition for Γ is

$$\Gamma = \{(V_1,V_2) : |V_1 \cap U| = m\}. \tag{3.7}$$

Of course, if $U = V$, we obtain again Γ_m. We can take one more step toward generality and consider a graph G with each vertex v has a weight $m(v)$. For example $m(v)$ could be the number of input bits read by mode v during the computation. Then we may set

$$\Gamma = \{(V_1,V_2) : \sum_{v \in V_1} m(v) = m\}. \tag{3.8}$$

Obviously 3.8 reduces to 3.7 when $m(v)=1$ for $v \in U$, and $m(v) = 0$ for $v \in V - U$. When dealing with a weighted graph it is more useful to include in Γ all dichotomies (V_1,V_2) such that V_1 has global weight in a given interval $[m_1,m_2]$. In fact we can state the following result.

Theorem 3.3. Let $G = (V,E)$ be a graph where each node v has a nonnegative integer weight $m(v)$. Let $M = \sum_{v \in V} m(v)$, and let $m(v) \leq m_2 - m_1 + 1$, for any v. If we define

$$\Gamma = \{(V_1,V_2) : m_1 \leq \sum_{v \in V_1} m(v) \leq m_2\} \tag{3.9}$$

then we have

$$A = \Omega(\frac{M}{m_2}\delta_\Gamma^2). \qquad 3.10$$

Proof. The argument is the same as the one made to prove Theorem 3.2, except that the claim made on the square cell of side $l \triangleq \left\lfloor (\delta_\Gamma - 1)/4 \right\rfloor$ will now state that the global weight of the nodes inside the cell is less than m_2. The bound $m(v) \leqslant m_2 - m_1 + 1$ ensures that if a cell has global weight m_2 or more, then it is always possible to construct a zig-zag cut delimiting a polygon with a perimeter $p \leqslant \delta_\Gamma - 1$, which includes a set of nodes with global weight in the interval $[m_1, m_2]$. Thus, less than δ_Γ edges connect the set V_1 of the nodes inside the polygon to the set V_2 of the nodes outside the polygon, contradicting the fact that $(V_1, V_2) \in \Gamma$. □

Remark. Theorem 3.2 is a special case of Theorem 3.3, and is obtained by setting $m_1 = m_2 = m$, and $m(v)=1$ for all v's.

3.2 INFORMATION EXCHANGE AREA-TIME LOWER BOUNDS (AT^2 THEORY)

In this section we introduce the notion of information exchange for a computational problem Π, and we relate it to the dichotomy width and to the AT^2 measure of computation graphs for Π.

Information Exchange. Let P_1 and P_2 be two processors cooperating to solve problem Π. Let $\mathcal{V}$ be the set of input and output variables of Π each of which is assumed to be binary. We call *I/O assignment* a partition $\eta = (\mathcal{V}_1, \mathcal{V}_2)$ of $\mathcal{V}$, where $\mathcal{V}_s$ is the set of variables that have to be input or output by processor P_s (s = 1,2). We define the *information exchange* of Π under assignment η as

$I(\eta) \triangleq$ the minimum over all the algorithms (that solve Π under the variable assignment η) of the maximum over all the problem instances of the number of bits exchanged between P_1 and P_2. 3.11

In other words, for any algorithm that solves Π under η there is at least a problem instance for which P_1 and P_2 exchange $I(\eta)$ or more bits, and no integer larger than $I(\eta)$ enjoys the same property.

We also define the information exchange for a class H of assignments as

$$I_H \triangleq \min_{\eta \in H} I(\eta). \qquad 3.12$$

Information and Dichotomy. Given a computation graph $G = (V, E)$ and a dichotomy $D = (V_1, V_2)$ of its nodes, we can identify P_s with the subgraph of G on vertex set V_s $(s = 1,2)$. This choice of P_1 and P_2 defines in a natural way an I/O assignment $\eta(D) = (\mathcal{V}_1, \mathcal{V}_2)$ where $\mathcal{V}_s$ is the set of variables input or output by nodes in V_s $(s = 1,2)$. We are then able to relate the notion of dichotomy width to that of information exchange.

Theorem 3.4. Let H be a class of I/O assignments for problem Π, with information exchange I_H. Let $G = (V, E)$ be a computation graph that solves Π in time T, and let δ_Γ be the dichotomy width of $\Gamma = \{D : \eta(D) \in H\}$. Then,

$$\delta_\Gamma \geq I_H / T. \qquad 3.13$$

Proof. If $D = (V_1, V_2) \in \Gamma$, then $\eta(D) \in H$, and $I(\eta(D)) \geq I_H$. Thus, V_1 must be able to exchange I_H bits with V_2 in time T, and therefore must be connected to V_2 by at least I_H / T edges. Hence, for each $D \in \Gamma$, $\delta(D) \geq I_H / T$, and $\delta_\Gamma = \min\{\delta(D) : D \in \Gamma\} \geq I_H / T$. □

AT^2 measure. We are now ready to state a result of major importance for the AT^2 theory.

Theorem 3.5. Given a computation graph G for problem Π, if the class $\Gamma = \{D : \eta(D) \in H\}$ generated by a class H of I/O assignments satisfies the conditions of Theorem 3.3, for a suitable choice of m_1, m_2 and of the weighting function $m(v)$, then the following lower bound holds on the area-time performance of G:

$$AT^2 = \Omega\left(\frac{M}{m_2} I_H^2\right). \qquad 3.14$$

Proof. It suffices to combine 3.13 and 3.10. □

The AT^2 lower bound 3.14 is a far reaching result because for many interesting computational prob-

lems we are able to (i) find a class H to which Theorem 3.5 is applicable, and (ii) compute or bound the information exchange I_H.

A Format for H. In several applications it is convenient to focus on a suitable set $\mathcal{U}$ of I/O variables, and to define H as

$$H = \{\eta = (\mathcal{V}_1, \mathcal{V}_2) : m_1 \leqslant |\mathcal{U} \cap \mathcal{V}_1| \leqslant m_2\}. \tag{3.15}$$

If we let $m(v)$ be the number of variables in $\mathcal{U}$ that are input or output by node v during the computation, then the class Γ of dichotomies associated to H is

$$\Gamma = \{(V_1, V_2) : m_1 \leqslant \sum_{v \in V_1} m(v) \leqslant m_2\}, \tag{3.16}$$

and, if $m(v) \leqslant m_2 - m_1 + 1$ all the conditions are satisfied for the validity of the bound $AT^2 = \Omega((M/m_2) I_H^2)$, where M is the total number of variables in $\mathcal{U}$. Thus, classes of I/O assignments of the kind specified by 3.15 are good candidates when studying the area-time complexity by means of Theorem 3.5. For this reason we further investigate the nature of I_H.

Some Properties of I_H. An interesting case of class H is obtained from 3.16 when $m_1 = m_2 = m$, i.e.

$$H_m \triangleq \{\eta = (\mathcal{V}_1, \mathcal{V}_2) : |\mathcal{U} \cap \mathcal{V}_1| = m\}. \tag{3.17}$$

In fact the classes H_m enable us to decompose H as

$$H = H_{m_1} \cup H_{m_1+1} \cup \cdots \cup H_{m_2}, \tag{3.18}$$

and, if we denote by $I(m)$ the information exchange of H_m, we can write

$$I_H = \min\{I(m_1), \ldots, I(m_2)\}. \tag{3.19}$$

A simple, but useful, observation is that, for any $m = 1, 2, \ldots, n$, we have

$$I(m) - I(m-1) \leqslant 1. \tag{3.20}$$

In fact, by just sending a bit from P_1 to P_2 [from P_2 to P_1], we can always transform an assignment

in H_m [H_{m-1}] into one in H_{m-1} [H_m]. Using the fact that $I(0)=0$ as the base, and Equation 3.20 as the inductive step of an inductive reasoning, we easily prove that $I(m) \leqslant m$. Another interesting consequence of 3.20 is that

$$I(m_2) - I_H \leqslant m_2 - m_1, \tag{3.21}$$

a result that may simplify the derivation of bounds on I_H.

A refinement on the AT^2 Bound. The fact that I_H is related to $I(m_2)$ by 3.21 suggests the possibility of obtaining a bound on AT^2 directly in terms of $I(m_2)$, a quantity easier to handle than I_H. Such a bound is indeed provided by the next theorem. As we shall see from the proof, the result is not trivial, and requires the combination of several arguments.

Theorem 3.6. Let G be a computation graph for problem Π. Let $\mathcal{U}$ be a set of I/O (binary) variables of Π, of cardinality M. If H_m is the class of the assignments such that exactly m variables of $\mathcal{U}$ are assigned to P_1, and $I(m)$ is the information exchange of H_m, then there exists a constant λ such that

$$AT^2 \geqslant \lambda\, M\, I^2(m)/m = \Omega(M\, I^2(m)/m). \tag{3.22}$$

Proof. Since a node v can read at most one (binary) variable per unit of time, $m(v) \leqslant T$, and condition $m(v) < m_2 - m_1$ is ensured by the choice $m_2 = m, m_1 = m - T$ in the definition 3.16 of H. With this choice, relations 3.13 and 3.21 imply that

$$I_H \geqslant I(m) - T,$$

and Theorem 3.5 (whose hypotheses are all satisfied) yields the bound

$$AT^2 \geqslant \lambda_1 W (I(m) - T)^2 / m, \tag{3.23}$$

for some constant λ_1. (If we retrace the proof of 3.23 we can see what $\lambda_1 = 1/25$ will do.)

When T approaches $I(m)$ from below, bound 3.23 may become weak, but because T is large we expect AT^2 to remain large. In fact

$$AT \geqslant (\textit{number variables be input or output by } G) \geqslant M,$$

and we have another bound

$$AT^2 \geq M\,T. \tag{3.24}$$

Combining 3.23 and 3.24 we obtain

$$AT^2 \geq \max\{\lambda_1 M\,(I(m)-T)^2/m, MT\}. \tag{3.25}$$

To prove that

$$\max\{\lambda_1 M\,(I(m)-T)^2/m, MT\} \geq \lambda_1 MI^2(m)/2m \tag{3.26}$$

we select for T the value

$$T_0 \triangleq I^2(m)/(2\,I(m)+\Lambda),$$

where $\Lambda = m/\lambda_1$, and we argue as follows.

(i) *For* $T \geq T_0$, $MT \geq \lambda_1 MI^2(m)/2m$. In fact $I(m) \leq m$, and λ_1 can be taken $\leq 1/2$, so that $2I(m)+\Lambda \leq 2\,\Lambda = 2m/\lambda_1$. Thus, $T_0 \geq I^2(m)/(2m/\lambda_1)$.

(ii) *For* $T \leq T_0$, $\lambda_1 M(I(m)-T)^2/m \geq \lambda_1 MI^2(m)/2m$. Since $T_0 < I(m)$ the function $(I(m)-T)^2/m$ in the interval $[0,T_0]$ is decreasing, and achieves its minimum at $T = T_0$. The value at the minimum is $(I(m)-T_0)^2/m = \dfrac{I^2(m)}{\Lambda}\dfrac{1/2+\Lambda}{4I+2\Lambda} \geq \dfrac{I^2(m)}{2\Lambda}$, which yields the desired result.

Equation 3.26 proves the theorem. Since $\lambda = \lambda_1/2$, λ can be taken to be 1/50. □

Remark. The value of T_0 used in the proof is an approximation of the (smallest) root of the equation in the unknown T obtained by equating the two bounds 3.24 and 3.25. The exact root would give a slightly better bound for λ , at the expense of more algebraic manipulations.

Remark. Although we have just proved that bound 3.22 holds for any T, the proof itself shows that, for $T > T_0$, the bound is weak, and that $AT \geq M$ provides more information on the area-time complexity of problem Π . However, when the computation is slow, the complexity is usually deter-

mined by other phenomena, as the ones to be discussed in the next section.

Boundary Chips. We briefly discuss now the situation where all the I/O ports are on the boundary of the layout region, which for simplicity we consider to be a rectangle R of dimensions l_x and l_y. In this case, as we have mentioned in Section 3.1, the I/O bounds requires $p = \Omega(M/T)$, where M is the input size. Thus, for the larger side of the rectangle, say, the horizontal side of length l_x, we have $l_x = \Omega(M/T)$. On the other hand, we have also seen in Theorem 3.3 that both l_x and l_y are at least $\delta(m)$ and we know that $\delta(m) \geq I(m)/T$. Thus, $A = l_x l_y = \Omega((M/T)(I(m)/T))$. In conclusion, the performance of boundary chips satisfies the bound

$$AT^2 = \Omega(M\ I(m)). \tag{3.27}$$

Remark. The value of m that yields the best bound in 3.27 is not necessarily the same that would give the best bound in 3.22.

3.3 SATURATION AREA-TIME LOWER BOUNDS

When we ideally isolate a region of the layout of a VLSI system, not only is the bandwidth between this region and the remaining part of the layout bounded by the perimeter of the region, but also the amount of information that can be stored within the region is bounded by its area. This fact has important consequences for the area-time performance of some computations. In this section we develop techniques to express these effects in a quantitative manner.

Information-Exchange Under Bounded Storage. We consider again the by now familiar framework in which two processors P_1 and P_2 cooperate to solve a given problem Π. However, we add a new element to the picture by assuming that only a limited amount of storage is available in each processor.

Storage limitations may affect the information exchange. In fact during the computation, one of the two processors may fill its storage (a situation referred to as "saturation") and hence be forced to send some information to its mate for temporary storage. At a later time, this information will return to the original processor, when its memory is no longer saturated. Each bit involved in this process goes

back and forth, contributing twice to the information exchange.

These considerations lead to the following formal definition of the information exchange of Π with respect to a given I/O assignment η under the condition that P_1 and P_2 can store at most s_1 and s_2 bits of information respectively.

$I(\eta \mid s_1, s_2) \triangleq$ the minimum over all algorithms (that solve Π under assignment η, and under storage bounds s_1 and s_2 for P_1 and P_2), of the maximum over all the problem instances of the number of bits exchanged between P_1 and P_2.

Similarly, the information exchange for a class H of assignments can be defined as

$$I_H(s_1,s_2) \triangleq \min_{\eta \in H} I(\eta \mid s_1,s_2) \qquad 3.29$$

Remark. The functions $I(\eta \mid s_1,s_2)$ and $I_H(s_1,s_2)$ are both nonincreasing in each of the two variables s_1 and s_2. (More storage never hurts.)

As in previous sections, of particular interest is the family of assignment classes defined with respect to a suitable set of I/O variables of the problem, that is

$$H_m = \{\eta = (\mathcal{V}_1,\mathcal{V}_2) : \mathcal{U} \cup \mathcal{V}_1 \mid = m\}. \qquad 3.30$$

For convenience of notation, we write $I(m \mid s_1,s_2)$ instead of $I_{H_m}(s_1,s_2)$.

The Square-Tessellation Technique. We now show that by combining bounds on the information exchange with bounded storage with the square-tessellation technique we can obtain area-time lower bounds.

We recall from Section 3.1 that a computation graph G is to be laid out on the layout grid, and that it is also useful to introduce the auxiliary grid whose vertices are the centers of the elementary cells of the layout grid.

Let us consider on the auxiliary grid a square cell with a side of length l as shown in Figure 3.5. We can identify the part of the graph laid out within the cell with processor P_1, and the part laid out

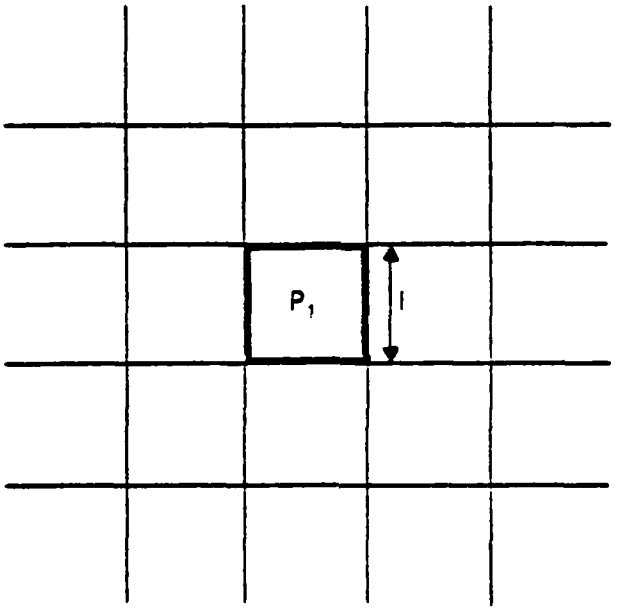


Figure 3.5. A cell of the square tessellation can be viewed as a processor P_1, with storage bounded by l^2. Identifying P_2 with the rest of the layout, the bandwidth between P_1 and P_2 is bounded by $4l$.

outside the cell with processor P_2. Obviously, the storage of P_1 is upper bounded by l^2. The storage of P_2 is also upper bounded by $A - l^2$, where A is the area of the layoug of the graph. However, in the sequel we will not make use of this bound.

If m variables of $\mathcal{U}$ are input or output (by nodes of the computation graph laid out) within the cell, then the information exchange across the boundary of the cell is at least

$$I(m, l^2, A - l^2) \geq I(m, l^2, +\infty).$$

Since the perimeter of the cell is $4l$, we can conclude that

$$T \geq I(m, l^2, +\infty)/4l. \qquad 3.31$$

Given a tessellation of the layout region with square cells of side l, we can argue that, since $M = \mathcal{U}$ variables are input or output in area A, there exists at least a cell C of the tessellation for which the number of variables of $\mathcal{U}$ handled by C is

$$m \geq Ml^2/A. \qquad 3.32$$

If we could find a cell of the tessellation for which m is exactly Ml^2/A , we could state the bound

$$T \geq I(Ml^2/A \mid l^2, \infty)/4l,$$

from Equation 3.31, which is indeed an area-time lower bound. A cell with m exactly equal to Ml^2/A does not always exist, but we can argue as follows to obtain a bound.

We know that, since at most T variables can be input or output by a node in time T, there exists a suitable zig-zag line that cuts the cell C into two regions one of which inputs and outputs a number $(Ml^2/A + h)$ of variables of $\mathcal{U}$, with $0 \leq h < T$. Moreover, the perimeter of this region is at most $4l$ and the area is at most l^2 . Thus, we can claim that an amount $I(Ml^2/A + h \mid l^2, \infty)$ of information must cross the boundary of this region. Hence,

$$T \geq I(Ml^2/A + h \mid l^2, \infty)/4l, \text{ for some } h \in [0, T-1]. \qquad 3.33$$

We can formally summarize the preceding discussion by stating the following theorem.

Theorem 3.7. Let G be a computation graph for problem Π . Let $\mathcal{U}$ be a set of I/O (binary) variables of Π , of cardinality M. If H_m is the class of assignments such that exactly m variables of $\mathcal{U}$ are assigned to P_1, and $I(m \mid s, \infty)$ is the information exchange of H_m when P_1 has s bits of storage, then the area-time performance of any layout of G satisfies the bound

$$T \geq \min_{0 \leq h \leq T} I(Ml^2/A + h \mid l^2, \infty)/4l. \qquad 3.34$$

Proof. Obvious from Equation 3.33.

Remark. To obtain the best possible bound from 3.34 we must choose the value of l that maximizes the right hand side. (We can choose l as we wish, since the inequality holds for arbitrary l .)

Remark. In most cases in the range of interest, $I(m \mid s, \infty)$ is increasing with m, so that the minimum in 3.34 is achieved for $h = 0$. Then we can state the bound as

$$T \geq \max I(Ml^2/A \mid l^2, \infty)/4l. \qquad 3.35$$

In all applications of 3.35 made in what follows (Theorems 4.5, 4.10, and 4.18) it turns out that, for the value l_0 of l that maximizes the lower bound, $I(m \mid l_0^2, \infty) = \beta_1 m$, where β_1 is a constant. Then we can write the bound as

$$AT \geq \beta Ml_0, \qquad 3.36$$

with $\beta = \beta_1/4$. Usually l_0 is an increasing function of the problem size, and therefore 3.36 is a better bound than the straightforward I/O bound $AT = \Omega(M)$.

CHAPTER 4

LOWER BOUNDS FOR CYCLIC SHIFT AND SORTING

4.1 CYCLIC SHIFT

Several lower-bound arguments for sorting are based on the fact that sorting circuits are capable of performing cyclic shifts on a suitable sequence of words. In this section we derive some results on the information exchange and the AT^2 measure for the cyclic shift problem. This will afford us the possibility to illustrate the lower bounds techniques of the preceding sections by applying them to a relatively simple problem.

Shift arguments have been first proposed by [BK80] and [AA80] to lower bound the AT^2 performance of integer multipliers, and they have also been successfully applied to sorting by [L84] whose results will be reviewed in Section 4.2.2.

Definition. The input of the (n,q)-cyclic shift problem is a pair (p,Z) where p is an integer between 0 and $n-1$, and

$$Z = \{Z^j_i : i = 0,1,\ldots,n-1; j = 0,1,\ldots,q-1\}$$

is an $n \times q$ array of n words of q symbols each. The output is an array W with the same format as Z, such that

$$W^j_i = Z^j_{(i-p) \bmod n}$$

In the following we shall assume that Z has binary entries. No assumption is made instead on the encoding used for the integer p, the size of the shift.

Information Exchange. With reference to the framework of Section 3.2, let $\eta = (\mathcal{V}_1, \mathcal{V}_2)$ be an I/O assignment for the (n,q)-cyclic-shift problem. We need to define a number of quantities that are functions of η:

$b_j \triangleq$ number of input words whose j-th bit is input by P_1,

$c_j \triangleq$ number of output words whose j-th bit is output by P_1,

$B \triangleq b_0 + b_1 + \cdots + b_{q-1}$ (global number of Z entries input by P_1),

$C \triangleq c_0 + c_1 + \cdots + c_{q-1}$ (global number of W entries output by P_1).

For a given shift p, it is easy to see that input position (i,j) contributes one bit to the information exchange if and only if Z_i^j and $W^j_{(i+p) \bmod n}$ are assigned to different processors. An immediate consequence is that

$$I(\eta) \geq | B - C |. \tag{4.1}$$

Let ϕ_j be the information exchange due to the j-th position summed over all the n different shifts. We claim that

$$\phi_j = b_j(n - c_j) + (n - b_j)c_j \tag{4.2}$$

In fact, each of the b_j bits input by P_1 is output by P_2 $(n - c_j)$ times, and, symmetrically, each of the $n - b_j$ bits input by P_2 is output by P_1 c_j times.

By the pidgeon-hole principle there is a shift size with information exchange not smaller than the average, which ensures that

$$I(\eta) \geq 1/n \sum_{j=0}^{q-1} \phi_j. \tag{4.3}$$

We can then derive bounds on $I(\eta)$ if we are able to bound the ϕ_j's. With the motivation that ϕ_j tends to be large when the output bits of position j are about equally split between P_1 and P_2, We classify the positions as follows. For given $\gamma \in [0,1/2]$ we define

$$Q_\gamma \triangleq \{ j : \gamma n \leq c_j \leq (1-\gamma)n \} \tag{4.4a}$$

$$Q_1 \triangleq \{ j : c_j > (1-\gamma)n \}, \tag{4.4b}$$

$$Q_2 \triangleq \{ j : c_j < \gamma n \}, \tag{4.4c}$$

$$q_s \triangleq |Q_s|, \; s = 0,1,2, \quad q_0 + q_1 + q_2 = q \tag{4.5}$$

$$B_s \triangleq \sum_{j \in Q_s} b_j, \; s = 0,1,2. \tag{4.6}$$

We can think of positions in Q_0 as "balanced", and positions in Q_s as P_s-biased, $s = 1,2$. We show now that the information exchange of a given assignment is at least proportional to the number of balanced positions.

Theorem 4.1. The information exchange of any assignment η for the (n,q)-cyclic-shift problem satisfies the inequality

$$I(\eta) > \gamma q_0 n. \tag{4.7}$$

Proof. If $j \in Q_0$, then both c_j and $n - c_j$ are $\geq \gamma n$, and

$$\phi_j = b_j(n - c_j) + (n - b_j)c_j \geq b_j \gamma n + (n - b_j)\gamma n = \gamma n^2. \tag{4.8}$$

Combining the last inequality with 4.3 we obtain

$$I(\eta) \geq (1/n) \sum_{j \in Q_0} \phi_j \geq (1/n) q_0 \gamma n^2 = \gamma q_0 n. \square$$

Theorem 4.1 implies that $q_0 < I/(\gamma n)$, and hence that $q_1 + q_2 \geq q - I/(\gamma n)$.

However, in some applications we need a bound on q_1 (or q_2) alone, which can be obtained in terms of the total number of inputs of P_1, as shown by the following theorem.

Theorem 4.2. The information exchange $I(\eta)$ of any assignment η of the (n,q)-cyclic-shift problem, such that P_1 reads exactly B entries of Z, satisfies the bound

$$I(\eta) > \gamma(B - B_1), \tag{4.9}$$

and, since $B_1 \leq n q_1$,

$$q_1 > (B - I(\eta)/\gamma)/n. \qquad 4.10$$

Proof. For $j \in Q_0 \cup Q_2$, $n - c_j \geq \gamma n$. Thus

$$\phi_j \geq b_j(n - c_j) \geq b_j \gamma n, \; j \in Q_0 \cup Q_2.$$

Then inequality 4.3 yields

$$I(\eta) \geq (1/n) \sum_{j=0}^{q-1} \phi_j \geq (1/n) \sum_{j \in Q_0 \cup Q_2} b_j \gamma n = \gamma(B_0 + B_2) = \gamma(B - B_1) \square$$

AT^2 Bounds. We begin with a simple but important result due to [BK80] and [AA80].

Theorem 4.3. For the *(n,1)*-cyclic-shift problem

$$AT^2 = \Omega(n^2). \qquad 4.11$$

Proof. From Equations 4.3 and 4.8, for $q = 1$, we obtain

$$I(\eta) \geq (b_0(n - c_0) + (n - b_0)c_0)/n.$$

If $H \triangleq \{\eta : b_0 = n/2\}$, then $I(\eta) \geq n/2$ for any $\eta \in H$. Then $I_H \geq n/2$, and the proof is completed by recalling from Theorem 3.6 that $AT^2 = \Omega(I_H^2)$. $\square$

If we try to extend the previous result to words of arbitrary length q, we immediately realize that bipartition techniques are not sufficient. For example, we can construct a balanced assignment η with $b_j = c_j = n$ for $j \leq q/2 - 1$, and $b_j = c_j = 0$ for $j \geq q/2$. $I(\eta)$ is clearly zero (if we neglect the information exchange related to the shift size). The point is that each bit position can be processed independently from the others, so that information exchanges remain confined to small sets of I/O variables. This is a typical situation in which the square tessellation technique reveals its effectiveness.

Theorem 4.4. For the (n,q)-cyclic shift problem

$$AT^2 = \Omega(q\, n^2). \qquad 4.12$$

Proof. Let $\mathcal{U} \triangleq \{Z_i^j : i = 0, \ldots, n-1; j = 0, \ldots, q-1\}$, and $M = |\mathcal{U}| = nq$. Let $H \triangleq \{\eta : P_1$ reads

exactly $n/2$ input variables }. We recall from Theorem 4.2 that, for $\gamma \in [0,1/2]$, $I(\eta) > \gamma(B - B_1)$, and we distinguish to cases:

(i) Q_1 is empty, so that, being $B_1 = 0$ and $B = n/2$, we obtain

$$I(\eta) > (\gamma/2)n.$$

(ii) Q_1 is not empty, so that at least for one j, $c_j > (1 - \gamma)n$. Thus $C \geq c_j > (1-\gamma)n$, and recalling 4.1 we obtain

$$I(\eta) \geq C - B > (1-\gamma)n - 1/2n = (1/2 - \gamma)n.$$

If we choose $\gamma = 1/3$, we see that in both cases $I(\eta) > n/6$. In conclusion $I_H > n/6$, and, from Theorem 3.6, with $M = nq$, $m = B = n/2$, and $I_H > n/6$, we obtain

$$AT^2 = \Omega\left(\frac{M}{m} I_H^2\right) = \Omega\left(\frac{nq}{(n/2)}\left(\frac{n}{6}\right)^2\right) = \Omega(qn^2) \ . \ \square$$

We will see next that, for suitably slow computations, the saturation technique allows us to derive better bound than that one provided by Theorem 4.4.

Information Exchange Under Bounded Storage. We consider now the case when P_1 has a storage capacity of s bits, and we show that, if $s < n$, the storage limitations really affect the information exchange. First, we need to prove a simple, but important lemma.

Lemma 4.1. In a cyclic shifter with a place-determinate I/O protocol all bits of Z^j must be input before any bit of W^j can be output.

Proof. For two arbitrary indices i_1 and i_2 $(0 \leq i_1 i_2 \leq n-1)$, we can find a suitable shift size such that $W_{i_2}^j = Z_{i_1}^j$. Thus, $W_{i_2}^j$ cannot be output before $Z_{i_1}^j$ is input.

Remark. A simple consequence of this lemma is that, for every j, there is a time t_j such that all bits of Z^j have been input, and no bit of W^j has been output. Then n bits describing Z^j are stored in the shifter at time t_j. It should be clear that these n bits need not necessarily be $Z_0^j, \ldots, Z_{n-1}^j$, for the

system is free to encode data arbitrarily for intermediate steps of the computation. However, since $Z_0^j,\ldots,Z_{n-1}^j$ are unrelated, any encoding of them requires at least n bits for a suitable value of the variables, i.e. for a suitable problem instance.

Theorem 4.5. Any (n,q)-cyclic shifter satisfies the bound

$$AT \geq \Omega(qn\sqrt{n}). \tag{4.13}$$

Proof. Let η be an I/O assignment such that P_1 outputs c_j bits of position W^j. If $c_j > s$, then at the instant t_j (when all inputs of Z^j have been input, but no output of W^j has been released) at least $c_j - s$ of the bits that have to be output by P_1 are stored in P_2. Eventually, these bits have to be transferred to P_1 in order to be output, thus cotributing an amount $(c_j - s)$ to the information exchange. If we let $(x)_+$ denote x when $x > 0$, and zero otherwise, we can write

$$I(\eta \mid s,\infty) \geq \sum_{j=0}^{q-1} (c_j - s)_+ \tag{4.14}$$

If we consider a value $s \leq (1-\gamma)n$, for some $\gamma \in [0,1/2]$, and we recall that $Q_1 = \{ j : c_j > (1-\gamma)n \}$, and that $q_1 = |Q_1|$ then Equation 4.14 easily yields

$$I(\eta \mid s,\infty) \geq \sum_{j \in Q_1} (c_j - s_j) \geq q_1((1-\gamma)n - s). \tag{4.15}$$

From Theorem 4.2, we also know that $I(\eta)$, and a fortiori $I(\eta \mid s,\infty)$, satisfies the bound

$$I(\eta \mid s,\infty) \geq \gamma(B - nq_1), \tag{4.16}$$

where B is the number of bits input by P_1 according to I/O assignment η. A linear combination of bounds 4.15 and 4.16 with coefficients γ and $(1-\gamma-s/n)$ respectively, yields

$$I(\eta \mid s,\infty) \geq \gamma(1-\gamma(1-s/n))B \tag{4.17}$$

where, as usual, $0 \leq \gamma \leq 1/2$ and $0 \leq s/n \leq 1-\gamma$. The best bound is obtained when $\gamma = (1-s/n)/2$ and is

system is free to encode data arbitrarily for intermediate steps of the computation. However, since $Z_0^j,\ldots,Z_{n-1}^j$ are unrelated, any encoding of them requires at least n bits for a suitable value of the variables, i.e. for a suitable problem instance.

Theorem 4.5. Any (n,q)-cyclic shifter satisfies the bound

$$AT \geq \Omega(qn\sqrt{n}). \tag{4.13}$$

Proof. Let η be an I/O assignment such that P_1 outputs c_j bits of position W^j. If $c_j > s$, then at the instant t_j (when all inputs of Z^j have been input, but no output of W^j has been released) at least $c_j - s$ of the bits that have to be output by P_1 are stored in P_2. Eventually, these bits have to be transferred to P_1 in order to be output, thus cotributing an amount $(c_j - s)$ to the information exchange. If we let $(x)_+$ denote x when $x > 0$, and zero otherwise, we can write

$$I(\eta \mid s,\infty) \geq \sum_{j=0}^{q-1} (c_j - s)_+ \tag{4.14}$$

If we consider a value $s \leq (1-\gamma)n$, for some $\gamma \in [0,1/2]$, and we recall that $Q_1 = \{ j : c_j > (1-\gamma)n \}$, and that $q_1 = |Q_1|$ then Equation 4.14 easily yields

$$I(\eta \mid s,\infty) \geq \sum_{j \in Q_1} (c_j - s_j) \geq q_1((1-\gamma)n - s). \tag{4.15}$$

From Theorem 4.2, we also know that $I(\eta)$, and a fortiori $I(\eta \mid s,\infty)$, satisfies the bound

$$I(\eta \mid s,\infty) \geq \gamma(B - nq_1), \tag{4.16}$$

where B is the number of bits input by P_1 according to I/O assignment η. A linear combination of bounds 4.15 and 4.16 with coefficients γ and $(1 - \gamma - s/n)$ respectively, yields

$$I(\eta \mid s,\infty) \geq \gamma(1 - \gamma(1 - s/n))B \tag{4.17}$$

where, as usual, $0 \leq \gamma \leq 1/2$ and $0 \leq s/n \leq 1-\gamma$. The best bound is obtained when $\gamma = (1 - s/n)/2$ and is

$$I(\eta \mid s,\infty) \geq \frac{1}{4}(1-s/n)B. \qquad 4.18$$

Then, by applying Theorem 3.7 to the class of assignments η such that P_1 inputs exactly B bits, and recalling that in our case $M = nq$, we obtain

$$T \geq \min_{0 \leq h < T} \frac{1}{4}(1-l^2/n)\left|\frac{nql^2}{A}+h\right|$$

and thus

$$T \geq \frac{1}{16}(1-l^2/n)lnq/A. \qquad 4.19$$

The best bound is obtained for $l = \sqrt{n/3}$ and is

$$AT \geq \beta_1 qn\sqrt{n} = \Omega(qn\sqrt{n}), \qquad 4.20$$

where $\beta_1 = 1/(24\sqrt{3})$. □

From Equations 4.20 and 4.12 we have seen that for any (n,q)-cyclic shifter, and constants β_1 and β_2, $AT \geq \beta_1 qn\sqrt{n}$, and $AT^2 \geq \beta_2 qn^2$. The latter bound is stronger for $T < (\beta_2/\beta_1)\sqrt{n}$, while the former is stronger for $T > (\beta_2/\beta_1)\sqrt{n}$. This fact indicates that the complexity of cyclic shift is dominated by pure information exchange for relatively fast computations, but is affected by storage limitations for slower computations.

4.2 SORTING.

We are finally ready to apply the general techniques described in the preceding sections to the derivation of area-time lower bounds for the sorting problem.

For the purposes of this section we classify sorting problems according to the relationship between the length k of the key and the number n of keys. There are three cases that need to be analyzed separately, and for which we introduce the following terminology:

short keys: $1 \leq k \leq \log n$ 4.21a

medium-length keys: $\log n < k < 2\log n$ 4.21b

long keys: $2\log n \leq k$. 4.21c

Between this classification of sorting problems, and the classification of multisets according to the range of optimality of different encoding schemes there is an intimate relationship, which will become more and more apparent as we proceed.

In each of the three cases defined above we will derive several lower bounds using different techniques. The bounds will be of the AT^2 type when the dichotomy or the square tessellation techniques are used in combination with the unconstrained information exchange, and will be of the AT type when the square tessellation technique is combined with the saturated information exchange.

The dichotomy technique gives satisfactory results only for keys of medium length, whereas the square tessellation technique yields the best bounds for short and long keys.

AT^2 and AT bounds complement each other in the sense that the former are better for $T < T_0$, and the latter are better for $T > T_2$, when T_0 is a suitable computation time for which the two bounds coincide.

Notation. We recall from Chapter 1 that the input of the (n,k)-sorting problem can be viewed as an $n \times k$ array of binary variables

$$X = \{X_{i,j} : i = 0,1,\ldots,n-1; j = k-1,k-2,\ldots,0\},$$

where $X_{i,j}$ is the coefficient of 2^j in the binary representation of the i-th input key. The i-th row of X, $X_{i,\cdot}$, represents the i-th input key, and the j-th column of X, $X_{\cdot,j}$, represents the j-th least significant position. A similar notation is adopted for the ouput array Y.

Remark. Here and hereafter n is generally assumed to be a power of 2. While this simplifies the treatment of several details, this assumption is not a serious restriction for asymptotic anlysis. In fact the complexity of sorting n keys (n being here an arbitrary integer) is never smaller than the complexity of sorting n_1 keys, where n_1 is the largest power of two not exceeding n, and it is never larger than the

complexity of sorting n_2 keys, where n_2 is the smallest power of two not smaller than n.

Finally, we recall that $r = 2^k$ is the cardinality of the set from which keys to be sorted are drawn.

4.2.1 Short Keys.

Let P_1 and P_2 be two processors cooperating in solving an (n,k)-sorting problem, with $k \leq \log n$. To give some intuition, let us consider the situation in which P_1 reads keys $X_0,\ldots,X_{n/2-1}$, and P_2 reads keys $X_{n/2},\ldots,X_{n-1}$, and let us try to estimate the information exchange I necessary to complete the sorting. We can argue as follows. From the analysis of the encoding of multisets developed in Section 2.2, we know that each processor can encode its input using $\theta(r\log(1+n/r))$ bits, and send the encoding to the other processor. Then each processor obtains complete information about the input, and can compute all the outputs it is required to produce without further communication with its mate. Thus, we can conclude that $I = O(r\log(1+n/r))$. It would not be difficult to prove that in this situation the outlined algorithm minimizes the information exchange, and that indeed $I = \theta(r\log(1+n/r))$. However, the class H of the assignments such that P_1 reads exactly $n/2$ input keys does not have the properties to guarantee $AT^2 = \Omega(I_n^2)$, because there might be no cut of the layout such that (nearly) half of the keys are input on either side of the cut, unless we assume a word-local protocol (see Section 1.1). In the next theorem we circumvent this difficulty by restricting our attention to the least significant bit position of the input. This choice is not random, and is suggested by the fact that insert-and-prune encodings allow the reconstruction of the entire input by looking only at the least significant bit position of the output (see Theorem 2.1). The following result has been obtained independently by [Sg84a].

Theorem 4.6. Any VLSI (n,k)-sorter, with $k \leq \log n$, satisfies the bound

$$AT^2 = \Omega(r^2\log^2(1+n/r)), \tag{4.22}$$

where $r = 2^k$.

Proof. With reference to the general framework of Section 3.2 let us consider the set $\mathcal{U} \triangleq \{X_i^0 : i = 0,1,\ldots,n-1\}$ of the bits in the least significant portion of the input keys. Let H be the class of I/O assignments such that exactly $n/2$ of the variables of $\mathcal{U}$ are read by P_1, and let I be the information exchange of H. Because of Theorem 3.6, to prove Equation 4.22 it is enough to show that $I = \Omega(r\log(1+n/r))$.

Given $\eta \in H$, we can assume, without loss of generality, that the $n/2$ members of $\mathcal{U}$ input by P_1 belong to keys $X_0, X_1, \ldots, X_{n/2-1}$. We also divide both the input and the output keys into $r/2$ segments of $2n/r$ consecutive words each (see Figure 4.1):

$$x\text{-}seg(h) \triangleq \{X_{h\,2n/r+q} : q = 0,1,\ldots,2n/r-1\}, \tag{4.23}$$

$$y\text{-}seg(h) \triangleq \{Y_{h\,2n/r+q} : q = 0,1,\ldots,2n/r-1\}, \tag{4.24}$$

for $h = 0,1,\ldots,\frac{r}{2}-1$. We say that *y-seg(h)* is P_s-biased ($s = 1,2$) if at least half of the l.s.b. of keys in the segment are output by P_s. There is one processor, say P_2, such that there are at least $r/4$ indices $h_0, h_1, \ldots, h_{r/4-1}$ for which *y-seg(h)* is P_2 - biased. Let $h_{r/4}, h_{r/4+1}, \ldots, h_{r/2-1}$ be the remaining indices.

We now construct a subproblem of sorting by setting all the bits of each input key, except the least significant one, to a constant value, such that

$$X_{p\,2n/r+q} = 2h_p + X^0_{p\,2n/r+q} \quad p = 0,1,\ldots,r/2-1,\ q = 0,1,\ldots,2n/r-1,$$

where $X^0_{p\,2n/r+q}$ is arbitrary. In the corresponding output, $y\text{-}seg(h_p)$ contains the sorted sequence of *x-seg(p)*, with the k-1 leading bits of each key representing h_p. The l.s. bits $Y^0_{n_p 2n/r}, \ldots, Y^0_{n_p 2n/r+2n/r-1}$ form a string of zeros followed by a string of consecutive ones. The number of zeros z_p obviously equals the number of variables $X^0_{p\,2n/r}, \ldots, X^0_{p\,2n/r+2n/r-1}$ which are zero. Thus, there are $2n/r+1$ possible outcomes for each *y-seg*. The situation is illustrated in Figure 4.1.

Let us now focus on values of h_p with $p < r/4$. All the l.s. bits of *x-seg(p)* are input by P_1, and at least n/r l.s.b. of $y\text{-}seg(h_p)$ are output by P_2. Thus, P_2 is capable to produce at least $n/r+1$ different outcomes, and therefore to distinguish among $n/r+1$ intervals in which z_p can fall. This is

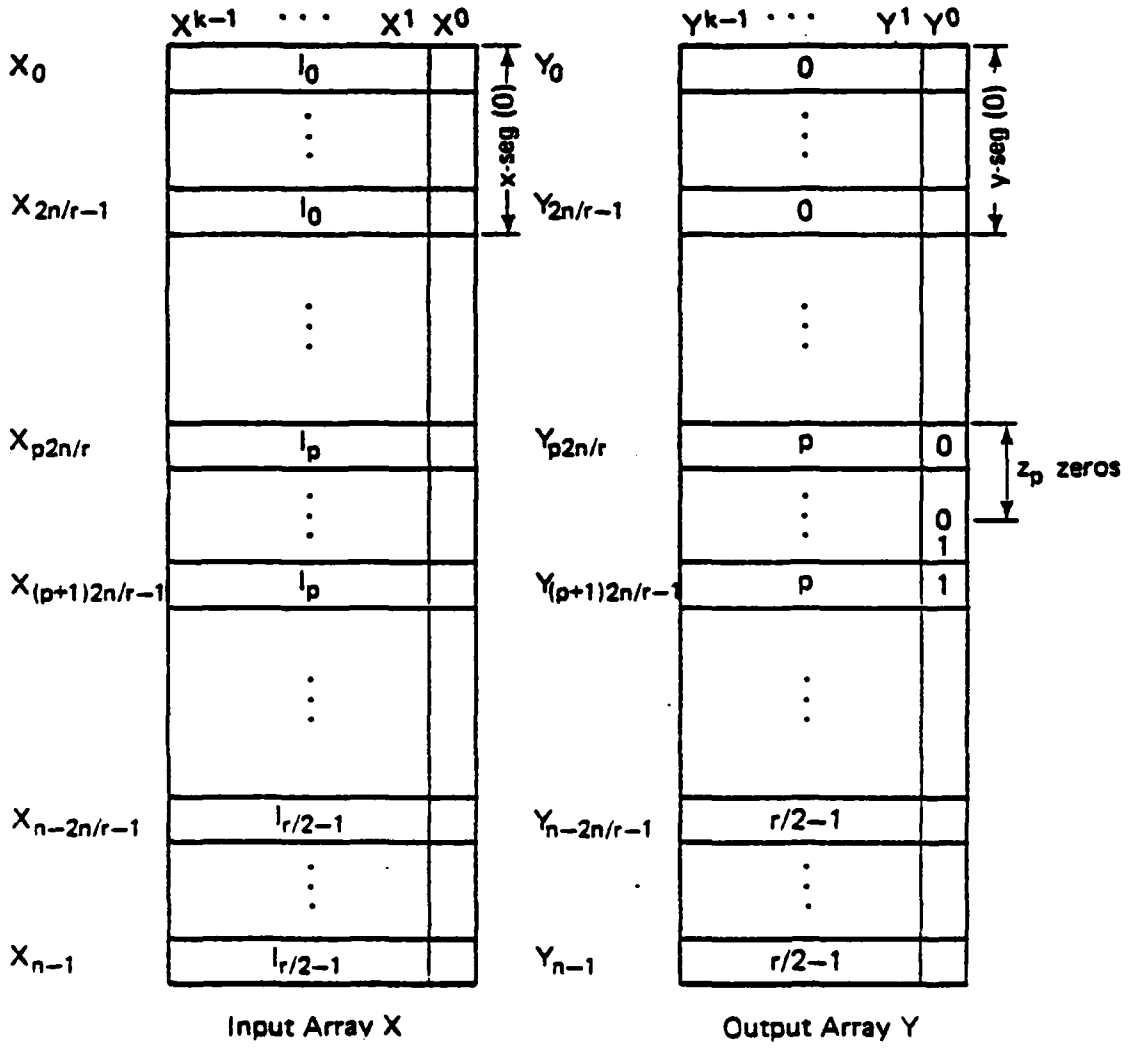


Figure 4.1. Input and output arrays in the proof of Theorem 4.6.

possible only if $log(n/r+1)$ bits relative to $x\text{-}seg(p)$ are communicated to P_2 and P_1. This being true for $r/4$ unrelated segments, we conclude as desired, that

$$I \geq r/4 \log(n/r + 1) = \Omega(r \log(n/r + 1)). \quad \square \qquad 4.25$$

If we combine the bound in Equation 4.25 with the bound in Equation 3.27 on the performance of boundary chips (in our case $M = nk$) we obtain the following result.

Theorem 4.7. Any VLSI (n,k)-sorter, with $k \leq \log n$, and with all its I/O ports on the boundary, satisfies the bound

$$AT^2 = \Omega(nkr\log(1 + n/r)) \qquad 4.26$$

where $r = 2^k$.

We have seen that for a protocol assigning half of the input keys to each processor the information exchange is $I = \theta(r\log(1 + n/r))$, but the situation can be dramatically different for other protocols, as illustrated by the next theorem.

Theorem 4.8. Given an (n,k)-sorting problem, with $k < \log n$, let H be the class of the I/O assignments such that P_1 inputs bit positions $X^0, X^1, \ldots, X^{k/2-1}$ and P_2 inputs bit positions $X^{k/2}, X^{k/2+1}, \ldots, X^{k-1}$ (k is even for simplicity). Then the information exchange of H is

$$I = \Omega(kn). \qquad 4.27$$

Proof. We plan to transform the string equality problem to our sorting problem. In the string equality problem there are two input strings, W_1 and W_2, of length h each, and there is one output bit which is 1 if and only if $W_1 = W_2$. It is easy to show that if processor P_s inputs string W_s, $s = 1,2$, then the solution of string equality requires an information exchange $I = h$ (see for example, [Y79], and [MS82]).

To carry out the transformation we set the first r input keys ($r < n$) to the constant value $X_i = i$, ($i = 0,1,\ldots,r-1$). From Theorem 2.1 we know that output position Y^0 is sufficient to reconstruct the entire input multiset.

Let us now define the strings W_1 and W_2. W_1 is the row-major spelling of the array

$$\{X_{i,j} : i = r, \ldots, n-1; j = k-1, \ldots, k/2\}.$$

W_2 is the row-major spelling of the array

$$\{X_i^j : i = r, \ldots, n-1;\ j = k/2-1, \ldots, 0\}.$$

(Refer to Figure 4.2). It is clear that $W_1 = W_2$ if and only if $X_i^j = X_i^{j+k/2}$ for $i = r, \ldots, n-1$, and $j =$

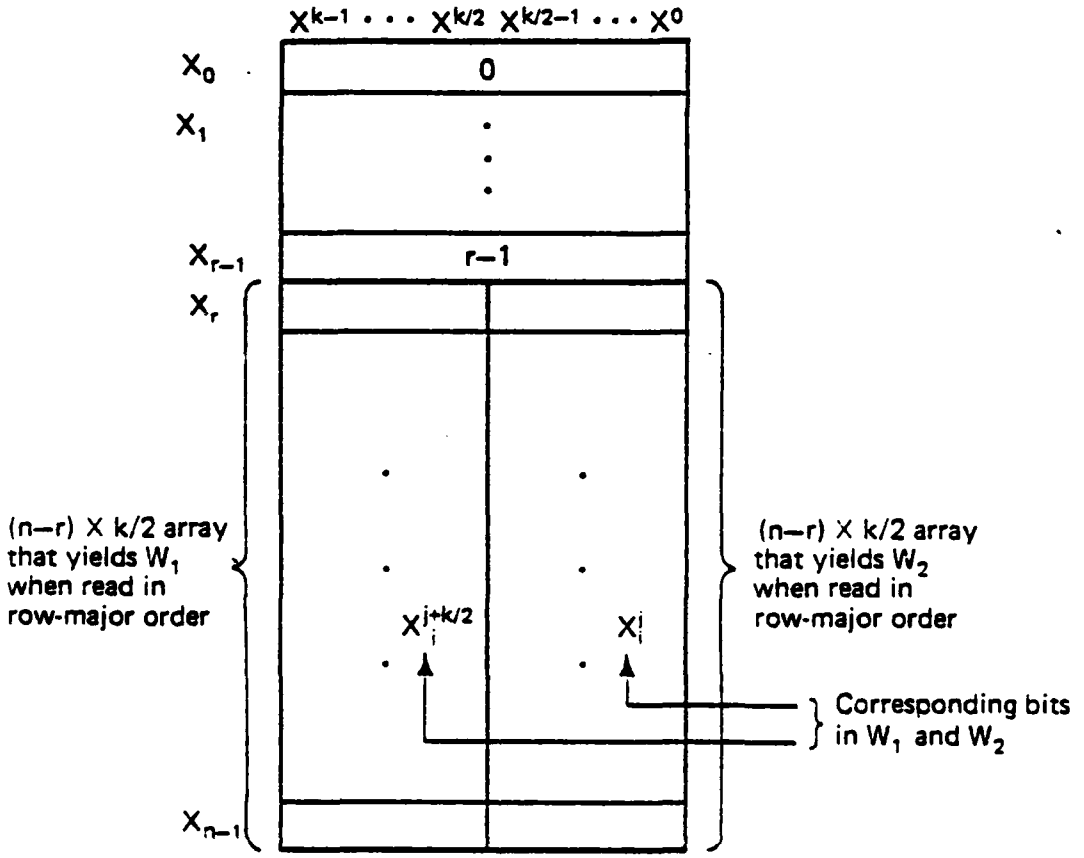


Figure 4.2. Configuration of the input array X in the proof of Theorem 4.8.

$0, \ldots, k/2-1$. This is equivalent to saying that each key X_i $(i \geq r)$ in the input multiset is the concatenation of two identical strings, a property of the multiset which is independent of its representation, and can therefore be verified once Y^0 is known. Now let P_s be the processor that outputs more variables of Y^0 (break a tie arbitrarily). Let P_s and P_{3-s} sort the input by exchanging I' bits, and let P_{3-s} receive $I'' \leq n/2$ bits to describe the components of Y^0 that it is required to output. With no further communication, P_s is now able to decide equality of W_1 and W_2, whose length is $h=(n-r)k/2$. Then $I' + I'' \geq (n-r)k/2$, and $I' \geq (n-r)k/2 - n/2 = \Omega(kn)$. □

The $AT^2 = \Omega(r^2\log^2(1 + n/r))$ obtained by bipartition techniques is weaker than the I/O bound $AT = \Omega(kn)$ for a wide range of values of r and T. However, we can greatly improve the AT^2 lower bound by the square tessellation method.

Theorem 4.9. Any VLSI (n,k)-sorter, with $k < \log n$, satisfies the bound

$$AT^2 = \Omega(nr), \tag{4.28}$$

where $r = 2^k$.

Proof. We plan to show that an I/O assignment in which P_1 reads exactly $r/2$ bits of the least significant input position requires an information exchange $\Omega(r)$. Equation 4.28 will then follow from Theorem 3.6 with $\mathcal{U} = \{X_i^0 : i = 0, \ldots, n-1\}$, $M = |\mathcal{U}| = n$, $m = r/2$, and $I = \Omega(r)$.

We begin by showing that, chosen an arbitrary set of input bits

$$\mathcal{U}_{in} = \{X_{s_0}^0, X_{s_1}^0, \ldots, X_{s_{r/2-1}}^0\} \tag{4.29}$$

and an arbitrary set of output bits

$$\mathcal{U}_{out} = \{Y_{t_0}^0, Y_{t_1}^0, \ldots, Y_{t_{r/2-1}}^0\} \tag{4.30}$$

with $t_i < t_{i+1}$, the remaining input bits can be set to constant values to enforce the condition

$$Y_{t_i}^0 = X_{s_i}^0 \quad i = 0,1,\ldots,r/2-1. \tag{4.31}$$

More specifically we set

$$X_{s_i} = 2i + X_{s_i}^0, \quad i = 0,1,\ldots,r/2-1,$$

and we divide the remaining n-r input keys arbitrarily into $r/2 + 1$ sets such that, for $i = 0,1,\ldots,r/2-1$, the i-th set contains $(t_i - t_{i-1} - 1)$ keys whose value is set to $2i$ $(t_1 \triangleq 0)$, and the $(r/2)$-th set contains $(n - 1 - t_{r/2-1})$ keys whose value is set to r-1. The output sequence corresponding to this input is shown in Figure 4.3, and it satisfies Equation 4.31.

Now, let us consider a protocol that assigns exactly $r/2$ variables of $\mathcal{U}$ to P_1 and $n - r/2$ $(\geq r/2)$ to P_2. Let P_s be the processor that outputs more entries of Y^0 (break a tie arbitrarily). We can always find two sets $\mathcal{U}_{in}$ and $\mathcal{U}_{out}$ as in 4.29 and 4.30 such that $\mathcal{U}_{in}$ is input by P_{3-s} and $\mathcal{U}_{out}$ is output by P_s. Equation 4.31 implies that $r/2$ bits input by P_{3-s} are output by P_s, for a suitable value of input variables not in $\mathcal{U}_{in}$. Hence, $I \geq r/2 = \Omega(r)$, as desired. □

We shall now prove an AT lower bound on the performance of an (n,k)-sorter for $k < \log n$. The proof is based on information exchange under bounded storage (saturation). However, the technique of Section 3.3 will be applied not to the entire computation interval $[0,T]$, but just to suitably defined subintervals.

Theorem 4.10. Any VLSI (n,k)-sorter, with $k \leq \log n$, satisfies the bound

$$AT = \Omega(n\sqrt{r}) \qquad 4.32$$

where $r = 2^k$.

Proof. For some real $\sigma \in [0, 1/2]$, in any tessellation of the layout with square cells of area σr, there is at least one cell C that outputs $m \geq n\sigma r/A$ bits belonging to Y^0, the least significant output position. Based on the output schedule of cell C, we partition the interval $[0,T]$ into consecutive intervals $[t_i + 1, t_{i+1}]$ (where $i = 0, 1, \ldots, L$-1, with $t_0 \triangleq -1$, and $t_L \triangleq T$), in such a way that, in each interval, C outputs between r/2 and $r(1/2 + \sigma)$ bits of Y^0. We can always find such a partition, since the cell can output at most σr bits at any given time. Furthermore, since cell C outputs $m \geq n\sigma r/A$ bits of Y_0, the number of intervals is at least $L \geq n\sigma/((1/2 + \sigma)A)$.

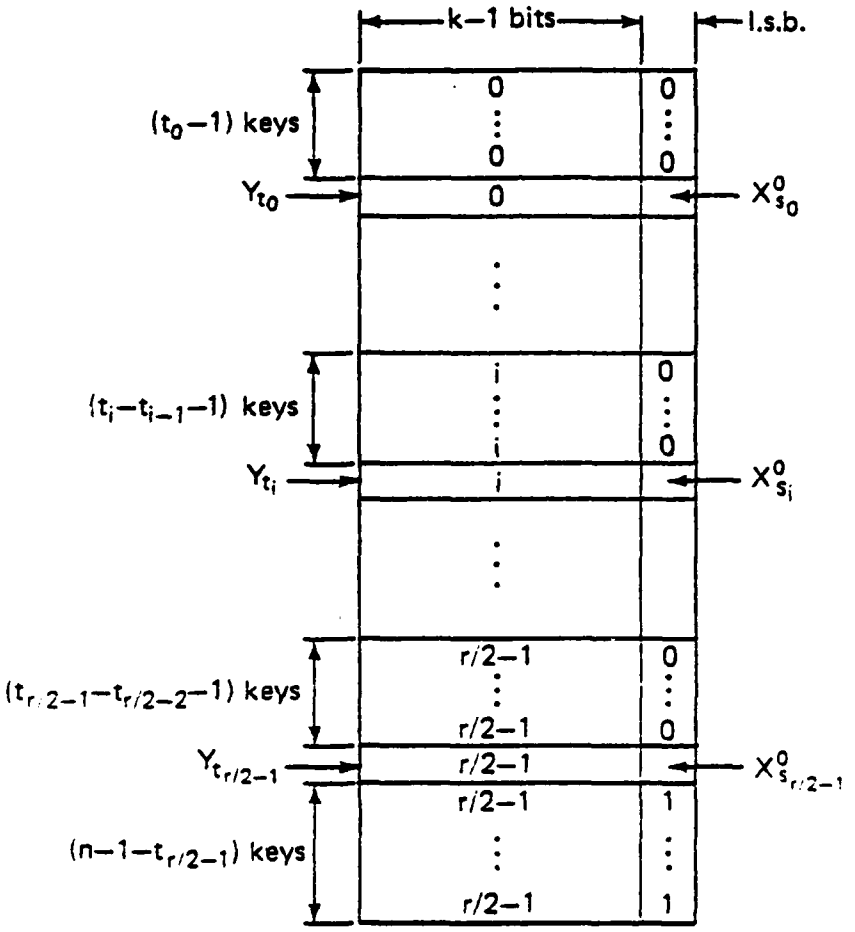


Figure 4.3. Sending $r/2$ arbitrary l.s. input bits to $r/2$ arbitrary l.s. output bits.

We now establish a lower bound on the duration $t_{i+1} - t_i$ of the i-th interval. As we have seen in Theorem 4.9, given an arbitrary sequence of $r/2$ components of Y^0, and an arbitrary sequence of $r/2$ components of X^0, it is possible to select the remaining inputs of the sorter in order to realize the identity function between the two sequences. Let us choose the $r/2$ components of Y^0 among those output

by cell C in $[t_i+1, t_{i+1}]$, and the $r/2$ components of X^0 in any arbitrary way. Then, since X^0 must be completely input before any bit of Y^0 can be output, during the interval $[t_i+1, t_{i+1}]$ cell C outputs $r/2$ bits that are already in the system at the beginning of the interval. Since C could store at most σr of them, the remaining $(1/2-\sigma)r$ must flow across the boundary of the cell, whose length is $4\sqrt{\sigma r}$, during the interval. Hence,

$$t_{i+1} - t_i \geq (1/2-\sigma)r / 4\sqrt{\sigma r} = \sqrt{r}\ (1/2-\sigma)/(4\sqrt{\sigma}), \tag{4.33}$$

and

$$T = \sum_{i=0}^{L-1}(t_{i+1}-t_i) + 1 \geq L\sqrt{r}\,(1/2-\sigma)/(4\sqrt{\sigma}). \tag{4.34}$$

Recalling the bound on L we obtain

$$AT \geq \frac{\sqrt{\sigma}}{4}\,\frac{1/2-\sigma}{1/2+\sigma} n\sqrt{r}, \tag{4.35}$$

which completes our proof. (Inequality 4.35 yields the best bound for $\sigma = (\sqrt{57}-7)/4 \simeq 0.138$). □

From Theorem 4.10 and Theorem 4.9 we know that there exist constants β_1 and β_2 such that the performance of any (n,k)-sorter, with $k \leq \log n$, satisfies the bounds $AT \geq \beta_1 n\sqrt{r}$, and $AT^2 \geq \beta_2 nr$. These bounds coincide at time $T_0 \triangleq (\beta_2/\beta_1)\sqrt{r}$. The AT bound is stronger for $T > T_0$, and the AT^2 bound is stronger for $T < T_0$.

The next two theorems provide us with some more information on the feasibility region of the sorting problem, for short keys.

The first theorem gives a lower bound on the area, regardless of the computation time. The same result has been independently obtained in [Sg84a] with a different proof.

The second theorem gives a lower bound on computation time, regardless of the area.

Theorem 4.11. The area of any VLSI (n,k)-sorter, with the $k \leq \log n$, satisfies the bound

$$A = \Omega(r \log(1 + n/r)). \tag{4.36}$$

Proof. As we have already seen, due to the functional dependence of the variables in Y^0 upon the

variables in X^0, and to the time-determinate property of the I/O protocol, that there is a time t^0 such that all the components of X^0 are input not later than t^0, and all the components of Y^0 are output after t^0. We have also seen that if we set the first r input keys to the constant value $X_i = i, i = 0, \ldots, r-1$, the remaining part of the input multiset $\{X_r, \ldots, X_{n-1}\}$ can be uniquely reconstructed from Y^0. Thus, a representation of $\{X_r, \ldots, X_{n-1}\}$ is essentially stored in the system at time t^0, and we know (see Eq. 2.13) that $\Omega(r \log(1 + n/r))$ bits are necessary to encode this multiset. □

Theorem 4.12. The computation time of any VLSI (nk)-sorter, with $k < \log n$, satisfies the bound

$$T = \Omega(\log n). \tag{4.37}$$

Proof. Equation 4.37 follows from the assumption of bounded fan-in when considering that the components of Y^0 depend on all the nk input variables. □

4.2.2 Medium-Length Keys

In this section we turn out attention to the $(n \log n + h)$-sorting problem, and we derive bounds for $0 < h < \log n$.

A simple observation, which is useful for lower bound arguments, is that by setting the $\log n$ leading bits of the input keys to an appropriate value, we can force the output sequence to be an arbitrary permutation of the input sequence. In particular the h least significant bits can be chosen arbitrarily to creat information flow.

This observation was originally exploited by Thompson [T80] to show that, for word-local protocols, and $k = \log n + \theta(\log n)$, $AT^2 = \Omega(n^2 \log^2 n)$. A straightforward generalization of Thompson's argument allows to prove the following theorem.

Theorem 4.13. Any VLSI $(n, \log n - h)$-sorter, with $h > 0$, and with word-local protocol, satisfies the bound

$$AT^2 = \Omega(n^2 h^2). \tag{4.38}$$

Proof. We will prove the theorem by showing that the class of I/O assignments $H = \{\eta : P_1 \text{ inputs exactly } n/2 \text{ keys}\}$ has information exchange $I = \Omega(n\,h)$.

Without loss of generality we can assume that P_1 inputs keys $X_0, \ldots, X_{n/2-1}$ and that P_2 outputs at least $n/2$ keys, say $Y_{a_0}, Y_{a_1}, \ldots, Y_{a_{n/2-1}}$. Let $Y_{a_{n/2}}, \ldots, Y_{a_{n-1}}$ be the remaining keys. By setting the $\log n$ leading bits of X_i to the binary representation of integer a_i we ensure that $Y_{a_i} = X_i$, $i = 0, \ldots, n-1$. Thus, the h least significant bits of each key input by P_1 are output by P_2, and $I \geq nh/2 = \Omega(nh)$, as claimed. □

No better bounds could be obtained on I under the word-local protocol, since $\theta(nk)$ bits are sufficient to encode the entire input. The important question instead is the removal of the "word-local" restriction.

Some preliminary considerations and an example will help us put in the proper perspective the nature of the problems arising when dealing with arbitrary protocols.

The output of the sorter is a permutation of the input, so that

$$Y_i = X_{\pi(i)} \qquad i = 0,1,\ldots,n-1 \tag{4.39}$$

where $\pi(0), \pi(1), \ldots, \pi(n-1)$ is a permutation of 0, 1, . . . , n-1. Focussing on the bit position of index j of the date we have

$$Y_i^j = X_{\pi(i)}^j \qquad i = 0,1,\ldots,n-1. \tag{4.40}$$

Thus, there is an information flow from the input to the output ports of the same position, which we call *primary flow*. The primary flow of each position is, in a way, self-contained, because each bit involved enters the system and leaves the system maintaining its identity. However, the exact destination of each bit within its own position depends on π, which, for position j, is determined by the value of the data in positions j, $j+1, \ldots, k-1$. Thus, there is another kind of information flowing from most significant to least significant positions, which we call *secondary flow*.

As we can see from the proof of Theorem 4.13, the complexity of word-local sorting is based exclusively on primary flow. Let us now consider an example of protocol which requires exclusively

secondary flow.

Example. We want to estimate the information exchange of the protocol that assigns the leading positions X^j, Y^j, $j = k/2, \ldots, k-1$ to P_1, and the least significant positions X^j, Y^j, $j = 0, \ldots, k/2-1$ to P_2. Since each bit position is completely input and output by the same processor, there is no primary flow. However, P_2 needs information on the relative order of the most significant part of the keys in order to know which permutation to apply to the least significant ones. Thus, we are in the presence of secondary flow alone.

It is clear that, no matter how large k is, $I \leq n\log n$. In fact the leading bits of P_1 can be sorted ignoring the bits of P_2, so that no information transfer is needed for P_2 to P_1. On the other hand, all that P_2 needs to know about the portion of keys dealt with by P_1 is the relative order, which can be encoded in no more than $n\log n$ bits.

We can also show that, for $k = 2\log n$, $I \geq \log(n!) = (n\log n$ - lower order terms). In fact, let us consider the class of instances of the problem such that $X_i = 2^{k/2}\pi(i) + i$ $(i = 0, \ldots, n-1)$, where $\pi(0), \ldots, \pi(n-1)$ is a permutation of $0, \ldots, n-1$. The corresponding output is $Y_i = 2^{k/2} i + \pi^{-1}(i)$ $(i = 0, \ldots, n-1)$. Thus, at least $\log n!$ bits (to describe π) are sent from P_1 to P_2. In conclusion, for $k = 2\log n$, $I = n\log n$-(lower order terms).

A more detailed analysis would show that, for $1 \leq k \leq 2\log n$, $I \approx nk/2$, and that for $k > 2\log n$ $I \approx n\log n$, regardless of k. The fact that secondary flow never exceeds $n\log n$ has important consequences, as we shall soon see. □

When analyzing arbitrary protocols, primary flow and secondary flow must be considered simultaneously. In fact, in particular situations one of the two may be negligible but, as we shall see, they cannot be simultaneously small.

Leighton [L84] has shown how to combine primary and secondary flow bounds with the help of cyclic shift arguments. His result was stated in the form $AT^2 = \Omega(n^2\log^2 n)$ for $k \geq 7\log n$.

Exploiting similar ideas, although with a rather more elaborate construction, we will show that $AT^2 = \Omega(n^2h^2)$ for $0 < h = k - \log n < \log n$. An obvious consequence is that for $k = (1+\alpha)\log n$, $(\alpha > 0)$, $AT^2 = \Omega(n^2\log^2 n)$. However, some discussion is in order, to clarify a subtle point. The Ω notation is misleading here (at least it has mislead us for some time), and it is better to rewrite the bounds in the following form : For $k = (1+\alpha)\log n$, $(\alpha > 0)$, $AT^2 \geq c(\alpha)n^2\log^2 n$, where $c(\alpha)$ depends on α, but is independent of n. The crucial point we want to address is that, for reasons that the next theorems on lower bounds will clarify and that are essentially related to the saturation behavior of secondary flow, the dependence of $c(\alpha)$ on α is quadratic for $\alpha << 1$, but is linear for $\alpha >> 1$. This fact, together with Leighton's observation that, when $k >> \log n$, one can construct VLSI sorters whose complexity is subquadratic in k [L84], shows that it would not be appropriate to consider a problem with $k = 1.1 \log n$ and a problem with $k = 100 \log n$ in the same class, although, superficially, we can say that $AT^2 = \Omega(n^2\log^2 n)$ in both cases.

We are then motivated to distinguish between medium-length and long keys. Obviously the choice of $k = 2 \log n$ as separation of the two classes is rather conventional, but it will serve our purpose. With this premise, we shall now prove the AT^2 bound for medium-length keys.

Theorem 4.14. Any VLSI $(n, \log n + h)$-sorter, with $0 < h < \log n$, satisfies the bound

$$AT^2 = \Omega(n^2h^2). \tag{4.41}$$

Proof. We begin by partitioning the input array as $X = [D, E, F]$ where D, E, F are blocks of d, $\log n - d$, and h consecutive columns respectively. The partition of the generic key X_i is shown in Figure 4.4.

We shall prove Eq. 4.41 by showing that $I = \Omega(n\,h)$ for the class of input assignments such that P_1 and P_2 input each exactly $nd/2$ of the entries of D. Below we shall derive two lower bounds on I, and we will see that at least one of these bounds is not smaller than $nh/12$.

Adopting the same notation as in Section 4.1 we let c_j be the number of components of Y^j out-

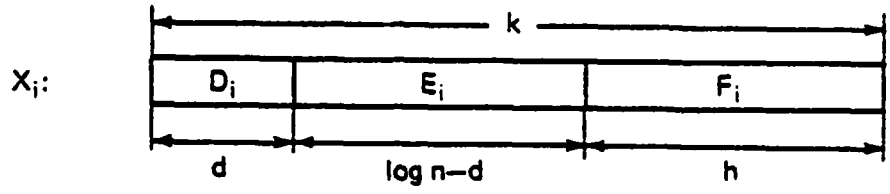


Figure 4.4. Partition of key X_i, for the proof of Theorem 4.14.

put by P_1. We also let, for $\gamma \in [0, 1/2]$, and $Q_0 \triangleq \{ j : j < h, \gamma n \leq c_j \leq (1-\gamma)n \}$, $Q_1 \triangleq \{ j : h, c_j > (1-\gamma)n \}$, and $Q_2 \triangleq \{ j : < h, c_j < \gamma n \}$. Finally we denote by q_i the cardinality of Q_i, $i = 0, 1, 2$.

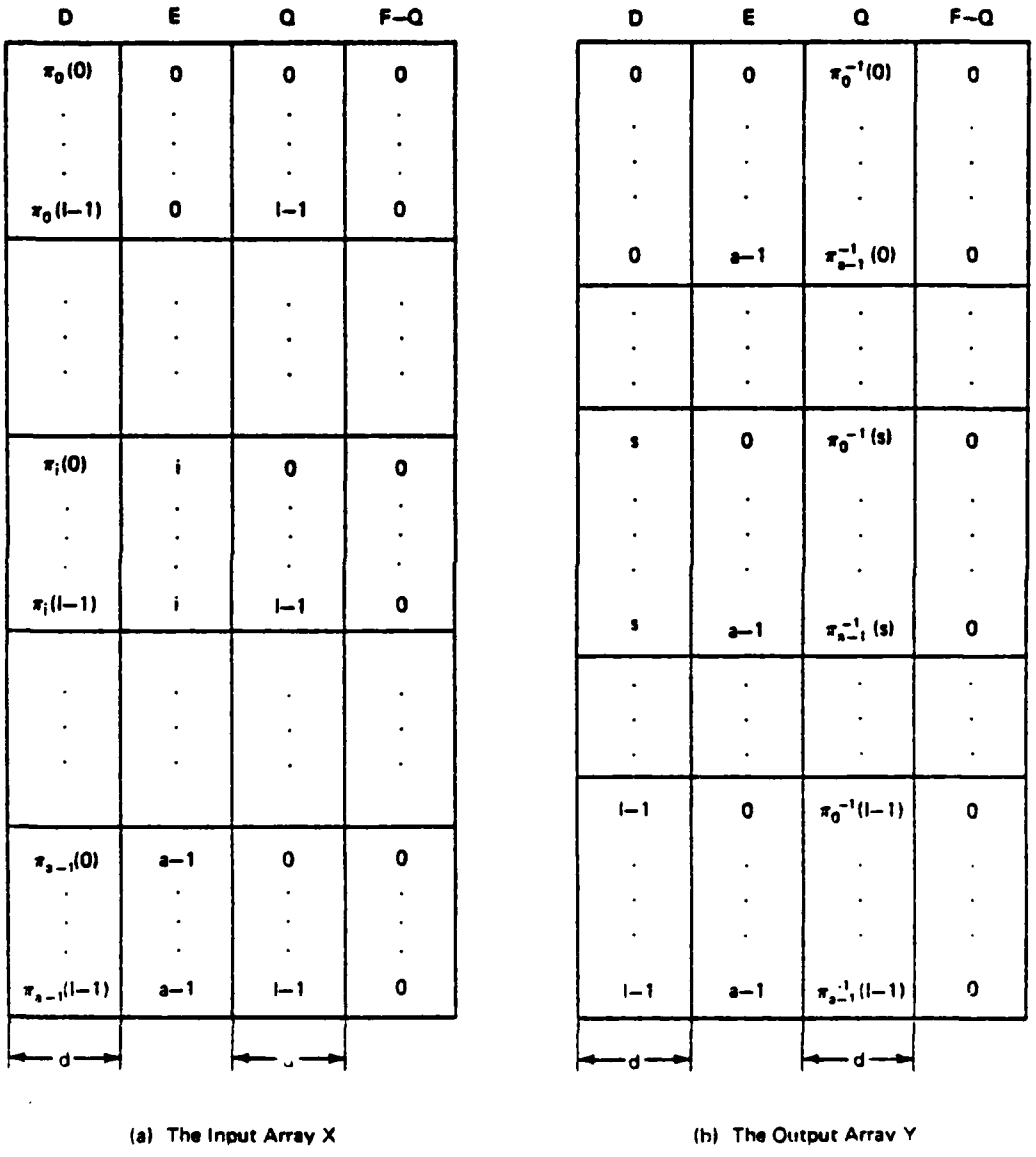
Primary Flow Bound. By suitably choosing the entries of D and E, we can produce any of the n cyclic shifts of array F. Then we can use the notation of Section 4.1 and Theorem 4.1, with $Z = F$, and $q = h$, to obtain the bound

$$l > \gamma q_0 n \qquad 4.42$$

which is valid for any $\gamma \in [0, 1/2]$, and where q_0 is a function of γ.

Secondary Flow Bound. Without loss of generality let $q_1 \geq q_2$, so that $q_1 \geq (h - q_0)/2$. Let also Q be a subset of Q_1 consisting of $d \geq q_1$ bit positions (the positions of Q are not necessarily consecutive.

but they will be thought of as ordered from most to least significant as they are in X). We then consider the following class of input instances, where $l \triangleq 2^d$ and $a = n/L$. Let input array X be partitioned into a blocks each of l consecutive rows, and let the entries of the $(i+1)$-st block $(i = 0, \ldots, a-1)$ be set so that: (see also Figure 4.5)



(a) The Input Array X

(b) The Output Array Y

Figure 4.5. Configuration of inputs and outputs in the proof of Theorem 4.14. In the actual arrays X and Y, the columns of blocks Q and F-Q are mixed, but the left-to-right order of the columns in each block is maintained.

(1) The rows of D have values $\pi_i(0),\ldots,\pi_i(l-1)$, where π_i is a permutation of 0, . . . , l-1.

(2) The rows of E are all identical and equal to i.

(3) The rows of Q have values 0, . . . , l-1.

(4) The rows of F - Q are set to zero.

It can be easily shown that if we partition the output array F into t blocks each of a consecutive rows, the $(s+1)$-st block from the top ($s = 0,\ldots,l-1$) has the following structure (see also Figure 4.5b):

(1) The rows of D have value s.

(2) The rows of E have value $0,\ldots,a-1$.

(3) The rows of Q have values $\pi_0^{-1}(s),\pi_1^{-1}(s),\ldots,\pi_{a-1}^{-1}(s)$.

(4) The rows of F - Q have value zero.

Thus, permutations $\pi_0,\pi_1,\ldots,\pi_{q-1}$ can be uniquely reconstructed from outputs in Q, so that Q carries $al\,logl$ - lower order terms) and bits of information relative to section D of the input. Since P_1 inputs only $nd/2$ of the bits describing $\pi_0,\ldots,\pi_{q-1}$, and outputs at least $(1-\gamma)nd$ of the bits of Q from which $\pi_0,\ldots,\pi_{q-1}$ can be recovered, we conclude that at least $(1-\gamma)nd - nd/2$ bits are transferred from P_2 to P_1. If we choose $d = q_1$, we obtain

$$I > (1/2-\gamma)q_1 n \geq (1/2-\gamma)[(h-q_0)/2]n, \qquad 4.43$$

where again $\gamma\in[0,1/2]$ and q_0 is a function of γ.

Combining the Bounds. If we select $\gamma = 1/6$, bounds 4.42 and 4.43 become

$$I > n\,q_0/6 \qquad 4.44$$

$$I > n\,(h-q_0)/6. \qquad 4.45$$

Thus,

$$I > \max(n\,q_0/6, n(h-q_0/6)) \geq nh/12. \qquad 4.46$$

(A slightly larger constant than 1/12 is obtained if we optimize the choice of γ, which yields $\gamma = (\sqrt{2}-1)/2$ and $I => nh/(1-\sqrt{2}/2)^2$. □

If we combine Inequality 4.46 with the bound in Equation 3.27 on the performance of boundary chips (in our case $M = \theta(n\log n)$) we obtain the following result.

Theorem 4.15. Any VLSI $(n,\log n+h)$-sorter, with $0<h<\log n$, and with all its I/O ports on the boundary satisfies the bound

$$AT^2 = \Omega(n^2 h \log n). \qquad 4.47$$

We end this section on medium-length keys with some results on minimum area and on minimum computation time. The result on the area (actually generalized to multilective I/O protocols), as well as the one on the AT^2-measure of Theorem 4.14 have been independently derived by [Sg84b], with a different approach.

Theorem 4.16. The area of any VLSI $(n\log n + h)$-sorter, with $0 < h < \log n$, satisfies the bound

$$A = \Omega(n\,h). \qquad 4.48$$

Proof. Due to the functional dependence of the variables in Y^j on the variables in X^i, with $j' \geq j$, and to the time- determinate property of the I/O protocol, there is a time t^* such that all the components of $X^{h-1}, X^{h-2}, \ldots, X^{h-\log n} = X^h$ are input not later than t^*, and all the components of $Y^{h-1}, Y^{h-2}, \ldots, Y^0$ are output after t^*.

Now, let us consider the same class of input instances as in Theorem 4.14, which is also illustrated in Figure 4.5a. At time t^* all entries of array D have been already input, and no entry of Q has been output yet. However, Q is an equivalent encoding of D, and hence $\Omega(a\ l\ \log l) = \Omega(n\ l)$ bits that represent D must be stored by the system at time t^*. □

Simple fan-in arguments allow us to prove the following result.

Theorem 4.17. The computation time of any VLSI $(n\log n + h)$-sorter with $0 < h < \log n$, satisfies the

bound

$$T = \Omega(\log n). \tag{4.49}$$

4.2.3 Long Keys

As we have anticipated in the preceding section, for $k > 2\log n$, bipartition techniques are not very useful, because there are input-balanced protocols for which the information exchange does not exceed $n\log n$, regardless of k.

However, we intuitively expect the area-time complexity of the (n,k)-sorting problem to be increasing with k, for fixed n. For example, the trivial I/O bound tells us that $AT = \Omega(k\,n)$. But we know more; in fact, a sorter of n keys of length k is trivially a cyclic shifter of n words of length $k-\log n$ (the least significant part of the keys), and hence it satisfies the bound $AT^2 = \Omega((k-\log n)\,n^2)$, according to Theorem 4.4.

This bound can be further improved by taking into account the fact that a suitable choice of the $\log n$ leading bit positions of the input keys of a sorter, can force at the output an arbitrary permutation of the keys, (not only the cyclic shifts).

Theorem 4.18. Any VLSI (n,k)-sorter, with $k \geq 2\log n$, satisfies the bound

$$AT^2 = \Omega\left(\left\lfloor \frac{k}{\log n}(n\log n)^2 \right\rfloor\right). \tag{4.50}$$

Proof. To simplify some details of the proof we assume $k \geq 3\log n$. (For $2\log n \leq k < 3\log n$ the result is a simple consequence of Theorem 4.14, in any case.) Since an appropriate selection of the $\log n$ leading bit positions of the input produces an arbitrary cyclic shift of the remaining positions, we can use some of the results derived in Section 4.1. Let us first recall some notations. We denote by b_j [respectively c_j] the number of input [respectively output] keys whose j-th bit is input [output] by P_i. Then for given $\gamma \in [0, 1/2]$, $Q_i \triangleq \{j : j < k - \log n, c_j < \gamma n\}$ and $q_i = |Q_i|$, $i = 1,2$. Moreover we let

$$B = \sum_{j=0}^{k-\log n-1} b_j, \text{ and } B_i = \sum_{j \in Q_i} b_j, \quad i = 1, 2.$$

The plan of the proof is to show that any I/O assignment in which P_1 reads $B = n\log n$ bits that belong to positions $X^{k-\log n-1}, \ldots, X^0$, requires an information exchange of $I = \Omega(n\log n)$ bits. Equation 4.50 will then follow from Theorem 3.6 with $\mathcal{U} \triangleq \{X_i^j : 0 \leq i \leq n-1,\ 0 \leq j \leq k - \log n - 1\}$, $M = |\mathcal{U}| = (k - \log n)n$, $m = B = n\log n$, and $I = \Omega(n\log n)$.

Primary Flow. By applying Inequality 4.9, and considering that, $\beta_1 \geq q_1 n$, and that in our case $B = n\log n$, we obtain

$$I \geq \gamma(n\log n - n\, q_1). \tag{4.51}$$

If we reverse the role of P_1 and P_2 in Theorem 4.52, and we consider that P_2 reads $(k - 2\log n)n \geq n\log n$ bits of $\mathcal{U}$, then Inequality 4.9 yields

$$I \geq \gamma(n\log n - n\, q_2). \tag{4.52}$$

Secondary Flow. Let P_s $(s = 1,2)$ be the processor that reads the majority of the bits that belong to the $\log n$ leading bit positions (break a tie arbitrarily). We will then show that the secondary flow increases with q_s. To be specific we will assume that $s = 2$, and we will bound the secondary flow from P_2 to P_1, which we finally combine with Eq. 4.51. (If $s = 1$, we can argue in a similar fashion resorting on Eq. 4.52.) After selecting arbitrarily a set Q^* of $(\log n - q_1)$ bit positions of significance less than $(k\text{-}\log n)$ and not in Q_1, we define the set

$$Q = Q_1 \cup Q^*.$$

We then consider the following class of input instances. We set:

(1) The leading $\log n$ bits of X_i to the value $\pi(i)$ where $\pi(i)$ is a permutation of $0, \ldots, n\text{-}1$.

(2) The $\log n$ bits of X_i which belong to positions in Q to the value i.

(3) All remaining bits to the value zero.

Then, the input array Y has the following structure.

(1) The leading $logn$ bits of Y_i represent integer i.

(2) The log$n$ bits of Y_i which belong to positions in Q represent integer $\pi^{-1}(i)$.

(3) All remaining bits are zero.

Thus, π can be recovered from the output positions $\{Y_j : j \in Q\}$. Since P_1 outputs at least $q_1(1-\gamma)n$ bits of these positions, and it reads at most $1/2\ nlogn$ bits among those that specify π, then

$$I \geq (1-\gamma)q_1 n - 1/2\, nlogn, \tag{4.53}$$

bits of information on π have to be communicated by P_2 to P_1.

Combining the Bounds. If we multiply both sides of bound 4.51 by $(1-\gamma)$, and both sides of bound 4.53 by γ, and we sum the sides of the resulting bounds, we obtain

$$I \geq \gamma(1/2-\gamma)nlogn. \tag{4.53}$$

For $\gamma = 1/4$, $I \geq nlogn/16$. Then, by Theorem 3.6 we have completed the proof. □

We derive now an AT bound for sorting of long keys, using saturation techniques.

Theorem 4.19. Any VLSI (n,k)-sorter, with $k \geq 2\,logn$ satisfies the bound

$$AT = \Omega(kn\sqrt{nlogn}). \tag{4.55}$$

Proof. In this proof we introduce several parameters whose value will be later specified to optimize the lower bound. The reader could find it useful to assume - in following the argument - that $\gamma = 1/12$, $\sigma = 5/24$, $\epsilon = 1/4$, $\xi = 1$, and $\beta = 3/8$. Although suboptimal, this choice of the parameters will give the right feeling for their range, and will also simplify the arithmetic.

We plan to lower bound the information exchange with bounded storage $I(m, s, \infty)$, and then apply Theorem 3.7. We consider the class of I/O assignments such that exactly m of the variables in $O = \{X_{ij} : 0 \leq i \leq n-1,\ 0 \leq j \leq k - logn - 1\}$ are input by P_1. As usual, we denote by c_j the

number of bits of position Y^j which are output by P_1, and - for $0 \leq \gamma \leq 1/2$ - we let $Q_1 = \{j : c_j > (1-\gamma)n, 0 \leq j \leq k - \log n - 1\}$, and $q_1 = |Q_1|$. As we have repeatedly seen, a trivial transformation of cyclic shift yields the following bound to the information exchange (under unbounded storage):

$$I > \gamma(m - n\, q_1). \qquad 4.56$$

Obviously, this bound holds a fortiori when the storage of P_1 is bounded, but we need to combine it with other bounds in order to obtain the desired result. The following observations provide some insight on how a bound on the storage may affect the information exchange.

(a) At the time when the last bit of a given position is input, no bits of that position have been output. Therefore n bits are stored in the system (P_1 and P_2) at that time.

(b) If during the time interval $[t_1, t_2]$ the system outputs p bits belonging to λ positions in set Q_1, then at least $p - \lambda\ \gamma\ n$ of those bits are output by P_1. (In fact, from the definition of Q_1, at most $\gamma\ n$ bits per position are output by P_2.)

(c) If, at a given time t, p bits that have to be output by P_1 are stored in the system, and P_1 has a storage bound of s bits, then at least $p - s$ bits are stored in P_2 at time t, and they will eventually be sent to P_1, thus contributing an amount p-s to the information exchange.

(d) If during the time interval $[t_1, t_2]$ P_1 outputs q bits which belong to a set $Q_1' \subseteq Q_1$ of $\lambda \leq \log n$ positions, then at least $(q$-$s)$ bits are transferred from P_2 to P_1 during the same interval, for an appropriate class of problem instances. The idea is that the outputs of P_1 carry q bits of information on the sorting permutation, and at most s of them could have been in P_1 at time t_1. The details of the argument are similar to those of the proof of Theorem 4.17. We need to set the $\log n$ leading bits of X_i to represent $\pi(i)$ (where $\pi(0), \ldots, \pi(n-1)$ is a permutation of $(0, \ldots, n\text{-}1)$. We also augment Q_1 to Q^* by adding ($\log n$ - λ) arbitrary positions, and we set the logn bits of X_i that belong to Q^* to the value i. Then the output position of Q^* will be $\pi^{-1}(0), \ldots, \pi^{-1}(n-1)$ where σ is the inverse of π and q bits of π are output by P_1.

In order to exploit the preceding observation systematically in the analysis of information exchange, we need to define several quantities. We begin by decomposing the interval $[0,T]$ during which the computation takes place, according to the I/O prorocol, which is assumed to be place-determinate and time-determinate.

If we focus on a given position j, we see that the variables of X^j are generally input at different times. We are particularly interested in the time when the ls. bit(s) of a given position are input. For our proof, we need to consider, for each time t, the number $\lambda(t)$ of positions in Q_1 whose last bit(s) are input exactly at time t. (For example, $\lambda(t)$ is zero when no bit is input at time t, but also when all the bits input at time t belong to positions for which some bits remain to be input.) We will treat separately the instants when $\lambda(t)$ is large from the instants when $\lambda(t)$ is small. In fact, in the first case we can immediately see that there must be a large saturation and secondary flow.

Formally, for a given ϵ $(0 \leqslant \epsilon \leqslant 1)$, we distinguish the times $t'_1 < t'_2 < \cdots < t'_u$ when $\lambda(t) \geqslant \epsilon \, logn$, from the times $t''_1 < t''_2 < \cdots < t''_v$ when $\lambda(t) < \epsilon \, logn$. Since for all the q_1 positions of Q_1 the input is completed at some time, we have

$$\sum_{h=1}^{u} \lambda(t'_h) + \sum_{h=1}^{v} \lambda(t''_h) = q_1. \qquad 4.57$$

We now consider separately the contribution to the information exchange due to positions whose input is completed in each of the two sequences. We assume that P_1 can store $s = \sigma \, nlogn$ bits $(0<\sigma<1)$.

Sequence t'. If we apply observation (a) to each of the $\lambda(t'_h)$ positions whose input is completed exactly at time t'_h, we see that at least $\lambda(t'_h)n$ bits are in the system at this time. From observation (b) at least $(1-\gamma)\lambda(t'_h)n$ of these bits have to be output by P_1. Finally, observation (c), with $t = t'_h$, $p = (1-\gamma)\lambda(t'_h)n$ and $s = \sigma \, nlogn$, allows the conclusion that the bit positions we are considering contribute $[(1-\gamma)\lambda(t'_h) - \sigma \, logn]n$ bits to the information exchange. If we sum over all t'_h we obtain a global contribution

$$I' = (1-\gamma)\sum_{h=1}^{u} \lambda(t'_h)n - u\,\sigma n logn. \qquad 4.58$$

Since $\lambda(t'_h) \geq \epsilon \log n$, we obtain

$$\sum_{h=1}^{u} \lambda(t'_h) \geq u \epsilon \log n,$$

and substituting for u in Eq. 4.58 we finally can write

$$I' > (1-\gamma-\sigma/\epsilon) \sum_{h=1}^{\nu} \lambda(t'_h) n. \qquad 4.59$$

Sequence t^*. We decompose the interval $[0,T]$ into consecutive intervals $[t^*_{h_i} + 1, t^*_{h_{i+1}}]$ for $i = 0, 1, \ldots, L$. (Here indices $h_1, \ldots, h_L \in \{0,1,\ldots,\nu\}$ and we have added to the sequence t^* two points, $t^*_{h_0} \triangleq -1$ and $t^*_{h_{L+1}} \triangleq T$.) The decomposition is chosen in such a way that, for a given ξ ($\epsilon \leq \xi \leq 1$) and for $i = 0, 1, \ldots, L$-1;

$$(\xi-\epsilon)\log n < \sum_{h=h_i+1}^{h_{i+1}} \lambda(t^*_h) \leq \xi \log n. \qquad 4.60$$

Such a decomposition always exists, since $\lambda(t^*_h) < \epsilon \log n$. Moreover,

$$L \geq \sum_{h=1}^{\nu} \lambda(t^*_h)/(\xi \log n), \qquad 4.61$$

since at most $\xi \log n$ positions complete their input in any given interval. We will evaluate the contribution to the information exchange given by each of the intervals of the decomposition. Let us focus on one specific time interval, say $[\tau_1,\tau_2]$. For a given β such that $0 < \beta < \xi - \epsilon$, we distinguish two cases.

(i) $p < \beta n \log n$. If $p < \beta n \log n$, at time t_2 at least $(\xi-\epsilon-\beta) n \log n$ bits are in the system, and at most $\gamma \xi n \log n$ of them have to be output by P_2. Thus, at least $((1-\gamma)\xi-\epsilon-\beta) n \log n$ bits have to be output by P_1, and, due to storage limitations, at least

$$I^*_0 \geq (\xi(1-\gamma)-\epsilon-\beta-\sigma) n \log n \qquad 4.62$$

of these bits are in P_2, and will eventually flow to P_1 contributing to the information exchange.

(ii) $p \geq \beta n \log n$. If $p \geq \beta n \log n$, at least $\beta n \log n - \xi n \log n$ of these bits are output by P_1, and observation (d) (with $q = (\beta-\gamma\xi) n \log n$, and $(\xi-\epsilon)\log n < \lambda \leq \xi \log n$) allows the con-

clusion that in the interval $[\tau_1,\tau_2]$ there is a contribution to the information exchange

$$I^*_1 \geq (\beta - \gamma\xi - \sigma)\, n\log n. \qquad 4.63$$

Now, if we chose $\beta = (\xi - \epsilon)/2$, then $I^*_0 = I^*_1$, and in either case the interval $[\tau_1,\tau_2]$ contributes

$$I^*_0 = I^*_1 = ((\xi - \epsilon)/2 - \gamma\xi - \sigma)\, n\log n \ \ bits. \qquad 4.64$$

Recalling Ineq. 4.61, we see that the global contribution of the L intervals of the decomposition is

$$I^* > ((1 - \epsilon/\xi)/2 - \gamma - \sigma/\xi) \sum_{h=1}^{L} \lambda(t^*_h)\, n. \qquad 4.65$$

If we chose $\xi = \epsilon(\sigma/\epsilon + 1/2)/(\sigma/\epsilon - 1/2)$, the coefficients of bounds 4.59 and 4.65 become equal, and by summing the contributions of the sequences of t'_s and t^*_s we obtain

$$I \geq (1 - \gamma - \sigma/\epsilon)\, n\, q_1 \qquad 4.66$$

where we have used Eq. 4.57. A linear combination of bounds 4.50 and 4.66 with coefficients $(1 - \gamma\,\sigma/\epsilon)$ and γ respectively, yields

$$I \geq \gamma(1 - \gamma/(1 - \sigma/\epsilon))\, m. \qquad 4.67$$

We now chose $\gamma = (1 - \sigma/\epsilon)/2$ to maximize the right hand side, so that 4.67 becomes

$$I(m \mid \sigma n\log n\ , \infty) \geq 1/4(1 - \sigma/\epsilon)\, m, \qquad 4.68$$

where we have used the appropriate notation for the information exchange. At this point we are ready to apply Theorem 3.7, which states that

$$T \geq I\,(Ml^2/A \mid h^2, \infty)/4l. \qquad 4.69$$

In our case $M = (k - \log n)/n$, and $l = \sqrt{\sigma\ n\log n}$, and we can rearrange bounds 4.69 and 4.68 as

$$AT \geq \frac{1}{16}\left|1 - \frac{\sigma}{\epsilon}\right| \sqrt{\sigma}(k - \log n)\, n\ \sqrt{n\log n}. \qquad 4.70$$

For a given σ, the best bound is obtained by maximizing ϵ. But ϵ is subject to the constraint $\xi = \epsilon(\sigma/\epsilon + 1/2)/(\sigma/\epsilon - 1/2) \leq 1$, so that $\sigma/\epsilon \leq (1 + \epsilon)/(2(1 - \epsilon))$. Under this constraint the lower bound on AT is maximized by the choice $\epsilon = 1/\sqrt{12}$, and

$\sigma = (\sqrt{12} + 1)/(2(12 + \sqrt{12}))$. □

From Theorem 4.19 and Theorem 4.18 we know that there exist constants β_1 and β_2 such that the performance of any (n,k)-sorter, with $k \geq 2\log n$ satisfies the bounds $AT \geq \beta_1 kn \sqrt{n\log n}$, and $AT^2 \geq \beta_2 kn (n\log n)$. These bounds coincide at time $T_0 = (\beta_2/\beta_1) n\log n$. The AT bound is stronger for $T > T_0$, and the AT^2 bound is stronger for $T < T_0$.

We complete the discussion of this section with two simple results on the minimum area and the minimum computation time for sorter of long keys. The first result has been originally proved by [L84], but it is also a trivial corollary of Theorem 4.16. The second result follows, as usual, by simple fan-in considerations.

Theorem 4.20. The area of any VLSI (n,k)-sorter, with $k \geq 2\log n$, satisfies the bound

$$A = \Omega(n\log n). \tag{4.71}$$

Theorem 4.21. The computation time of any VLSI (n,k)-sorter, with $k \geq 2\log n$, satisfies the bound

$$T = \Omega(\log n + \log k). \tag{4.72}$$

Remark. Bound 4.72 is indeed satisfied by all sorters, but the dependence on k becomes relevant only for very long words, i.e., when $\log k = \Omega(\log n)$. We conclude Section 4.2 summarizing the main results in Table 4.1.

4.3 AREA-TIME LOWER BOUNDS FOR THE COMPARATOR-EXCHANGER

Usually *comparison-exchange* is formulated as a problem whose input consists of two keys X_0 and X_1, and whose output consists of two keys Y_0 and Y_1 such that

$$Y_0 = \min(X_0, X_1), \tag{4.73}$$

$$Y_1 = \max(X_0, X_1). \tag{4.74}$$

Comparison-exchange is an interesting operation in its own right, and it is a primitive of many sorting

TABLE 4.1. SUMMARY OF LOWER BOUNDS FOR (n,k)-SORTING

Length of the Keys / Lower Bound Techniques	$1 \leq k \leq \log n$ $(r = 2^k)$	$\log n < k < 2\log n$ $(h = k - \log n)$	$2\log n \leq k$
Bipartition + Information Exchange	$AT^2 = \Omega(r^2 \log^2(1+n/r))$	$AT^2 = \Omega(n^2 h^2)$	$AT^2 = \Omega(n^2 \log^2 n)$
Square Tesselation + Information Exchange	$AT^2 = \Omega(nr)$	—	$AT^2 = \Omega(nk(n\log n))$
Square Tesselation + Saturated Information Exchange	$AT = \Omega(n\sqrt{r})$	—	$AT = \Omega(nk\sqrt{n\log n})$
Storage	$A = \Omega(r\log(1+n/r))$	$A = \Omega(nh)$	$A = \Omega(n\log n)$
Bounded Fan-in	$T = \Omega(\log n)$	$T = \Omega(\log n)$	$T = \Omega(\log n + \log k)$

algorithms. However, our main motivation to study its area-time complexity comes from the fact that comparison-exhcnage is indeed the $(2,k)$-sorting problem, and its analysis will provide us with useful insight into the phenomena that determine the complexity of sorting when the length k of the keys is very large with respect to their number n.

The lower bound technique that we shall adopt is different from the ones we have applied in the preceding sections, and it is based on the notion of functional dependence.

The notion of functional dependence has been introduced in the context of VLSI computation by Johnson [Jh80]. In order to derive an area-time lower bound for binary addition, a problem similar to comparison-exchange in several respects.

Let us now recall the formal definition of functional dependence.

Definition. Given a function $\mathbf{y} = \mathrm{f}(\mathbf{x})$, where $\mathbf{x} = (x_1,\ldots,x_p)$ and $\mathbf{y} = (y_1,\ldots,y_q)$ are boolean vectors, we say that y_j is *functionally dependent* on x_i if there exist two boolean vectors $\mathbf{x}'$ and $\mathbf{x}''$ that differ only in the i-th component, such that $\mathbf{y}'=\mathrm{f}(\mathbf{x}')$, and $\mathbf{y}''=\mathrm{f}(\mathbf{x}'')$ differ in the j-th component.

Example. In the comparison-exchange problem, $Y_{i_0}^{j_0}$ is functionally dependent on X_i^j for any $j \geq j_0$, and $i = 0, 1$; $Y_{i_0}^{j_0}$ is *not* functionally dependent on X_i^j for any $j < j_0$, and $i = 0, 1$.

Time-Determinacy. For time-determinate protocols the functional dependence of y_j on x_i implies that x_i must be input before y_j is output, because there are input instances in which y_j remains indeterminate until x_i is known. However, more complex phenomena take place when we bring into the picture the asumption of bounded fan-in, which was not needed when analyzing the aspects of the computation that depend only on information exchange.

Bounded Fan-In. We explicitly assume now that in our circuits the gates that compute boolean functions have a number of input lines upper-bounded by a constant f_I. As is well known, this assumption implies that if an output variable y is functionally dependent on s input variables, then at least $\tau \log_{f_I} s$ time must elapse between the instant when the first of the input variables is read, and the instant when y is output, where τ is the minimum delay of a boolean gate, and f_I is the maximum fan-in. Hereafter, since the value of τ and f_I affects only constant factors, we assume for simplicity that $\tau = 1$ and $f_I = 2$.

Computational Friction. Although the previous considerations are often useful to bound the computation time of some problem, they do not exhaust all the consequences of functional dependence. In fact, if s variables $x_1,\ldots,x_s$, are input at the same time, and if there happen to exist s output variables $y_1,\ldots,y_s$ such that, not only each y_j depends on all s_i's but also the y's carry I bits of information on the x's, the system must be capable of storing I bits for at least $\log s$ time steps. Thus, if we make an analogy in which the information is viewed as a fluid flowing from input ports to output ports, we can

say that the functional dependence acts as a kind of friction that slows down the flow, keeping it below capacity.

In a VLSI system the I/O capacity is determined by the area, and implies the trivial I/O bound $AT = \Omega$ (*input size + output size*), already discussed in Section 3.1. When functional dependence plays a role in slowing down the I/O information flow, we intuitively expect the AT measure to satisfy a stronger lower bound.

This in indeed the case for comparison-exchanger, as shown in the following theorem.

Theorem 4.22. Any comparator-exchange of keys of length k satisfies the bound

$$A = \Omega((k/T)\log(k/T)), \tag{4.75}$$

which can be also rewritten as

$$AT/\log A = \Omega(k). \tag{4.76}$$

Proof. For $t = 1, 2, \ldots, T$, let $S(t)$ be the set of bits of X_0 and X_1 that are input exactly at time t, and let $s(t) \triangleq |S(t)|$. We partition $S(t)$ into two subsets $S_0(t)$ and $S_1(t)$ of equal size $s(t)/2$, the significance of the bits of $S_1(t)$. We consider now the set $C_0(t)$ containing all the output bits that belong to a position j such that at least one of X_0^j and X_1^j is input exactly at time t. Formally,

$$C_0(t) \triangleq \{Y_0^j : X_0^j \in S_0(t) \text{ or } X_1^j \in S_0(t)\} \cup \{Y_1^j : X_0^j \in S_0(t) \text{ or } X_1^j \in S_0(t)\}.$$

On set $C_0(t)$ we can make two important observations:

(i) All variables in $C_0(t)$ are functionally dependent on all variables in $S_1(t)$. Therefore, no variable in $C_0(t)$ can be output before time $t + \log(s(t)/2)$.

(ii) From the value of the variables in $C_0(t)$ - possibly with the addition of one extra bit specifying whether X_0 or X_1 is the smaller key - we can uniquely reconstruct the value of the variables in $S_0(t)$. Therefore, from time t to the time when the first variable of $C_0(t)$ is output, at least $s(t)/2$ bits of information concerning $S_0(t)$ must be stored by the system.

Combining these two observations, we conclude that the $s(t)$ variables input exactly at time t give a contribution of at least $(s(t)/2)\log(s(t)/2)$ *bit* $\times$ *time unit* to the *storage* $\times$ *time* product, and hence to the AT product. Thus,

$$AT \geq \sum_{t=1}^{T} s(t)/2 \log(s(t)/2), \qquad 4.77$$

where obviously $\sum_{t=1}^{T} s(t) = 2k$. Under this constraint, the right hand side of 4.77 is minimized when $s(t) = 2k/T$ for each $t = 1, \ldots, T$. Thus,

$$AT \geq k \log(k/T) \qquad 4.78$$

which proves 4.75. This can be also rewritten as 4.76 after simple algebraic manipulations. □

It is worth comparing Theorem 4.22 with Johnson's work [J80] from which we have borrowed the main idea, in order to clarify two superficial differences. First, although the area-time complexity of binary addition is exactly the same as the complexity of comparison-exchange, [J80] states the lower bounds in a form different from (and probably less clear than) Equations 4.75 and 4.76, in an attempt to formulate the results as a bound on a measure of the AT^{α} type. Second, while in our proof we bound essentially the amount of time that the input information must spend inside the system, Johnson bounds from below the duration of intervals during which a given amount of information is output by the system. However, this difference is only superficial, because it is obvious that when the storage has been saturated by inputs on which the system is still performing some computation, both the input flow and the output flow must necessarily slow down.

PART II

UPPER BOUNDS

CHAPTER 5

ALGORITHMS AND ARCHITECTURES

5.1 INTRODUCTION

In Part II of this thesis we turn our attention to the design of VLSI sorting circuits. A VLSI design has two fundamental aspects: the algorithm and the architecture. Both aspects have been extensively investigated by many researchers, and the valuable knowledge that has been accumulated is very useful for our study of VLSI sorting.

In Chapter 5 we review known parallel sorting algorithms and known parallel architectures that will be basic ingredients of our sorters. An effort has been made to give a unified presentation of the subject, but there is no attempt to make an exhaustive survey. Only algorithms and architectures that we actually use in subsequent chapters are indeed described.

In Chapter 6 we propose a variety of optimal sorters for keys of length $k = \log n + \theta(\log n)$. The designs are all based on previously known algorithms or simple variants thereof, and their novelty consists in the development of the appropriate architectures amenable to compact layouts.

In Chapter 7 we consider arbitrary key lengths and propose optimal or near-optimal designs. Several algorithms are new, and in fact it can be proven that for certain ranges of key lengths none of the classical sorting algorithms can achieve area-time optimality.

The performance of the proposed design is contrasted with the appropriate lower bounds. However, the presentational subdivision into lower and upper bounds, which considerably simplifies the exposition, does not reflect the real development of the problem analysis. An attempt to better relate lower bounds and upper bounds is made in Chapter 8 where the main results of the entire thesis are summarized and compared with each other.

5.2 PARALLEL ALGORITHMS FOR SORTING

In this section we review some known algorithms for parallel sorting, which will be implemented in the design of VLSI sorters.

5.2.1 The Combination Scheme

Several sorting algorithms can be viewed as particular cases of a rather general scheme, which we now describe.

We call *combination* the operation that produces from m sorted sequences of l elements each one sorted sequence of ml elements. A network implementing this operation is called an (m,l)-combiner. When $m = 2$, combination reduces to merging.

Given $n = m_1 m_2 \dots m_2$ elements, we can sort them in d stages according to the following scheme that we call combine-sort.

At stage 1 we perform n/m_1 combination operations, each on m_1 sequences of 1 element each. At stage 2 we perform $n/m_1 m_2$ combinations, each on m_2 sequences of m_1 elements each, and at stage i we perform $n/m_1 \dots m_i$ combination, each on m_i sequences of length $m_1 \dots m_{i-1}$. Finally, at stage d we combine m_d sequences of length n/m_d into one sequence of length n, which is the output of the combine-sort scheme. A diagrammatic illustration of the scheme is given in Figure 5.1 in the form of a rooted tree. Each node of this tree is a suitable combiner. An (m_i, l_{i-1})-combiner, $1 \leqslant i \leqslant d$, performs the combination of m_i (sorted) sequences of length l_{i-1}; here $l_0 = 1$ and $l_{i-1} \triangleq m_1 m_2 \cdots m_{i-1}$ for $i > 1$. Note that each level of the tree corresponds to a stage of the combination scheme, and that there are $n_i \triangleq n/l_i$ nodes at level i, $1 \leqslant i \leqslant d$.

Several known sorting algorithms can be cast in the combine-sort scheme. Each algorithm is characterized by a particular factorization of $n = m_1 \dots m_d$ (note that the order of the factors is relevant here), and by the specification of how the combination is to be performed.

We shall discuss some important algorithms in the following sections.

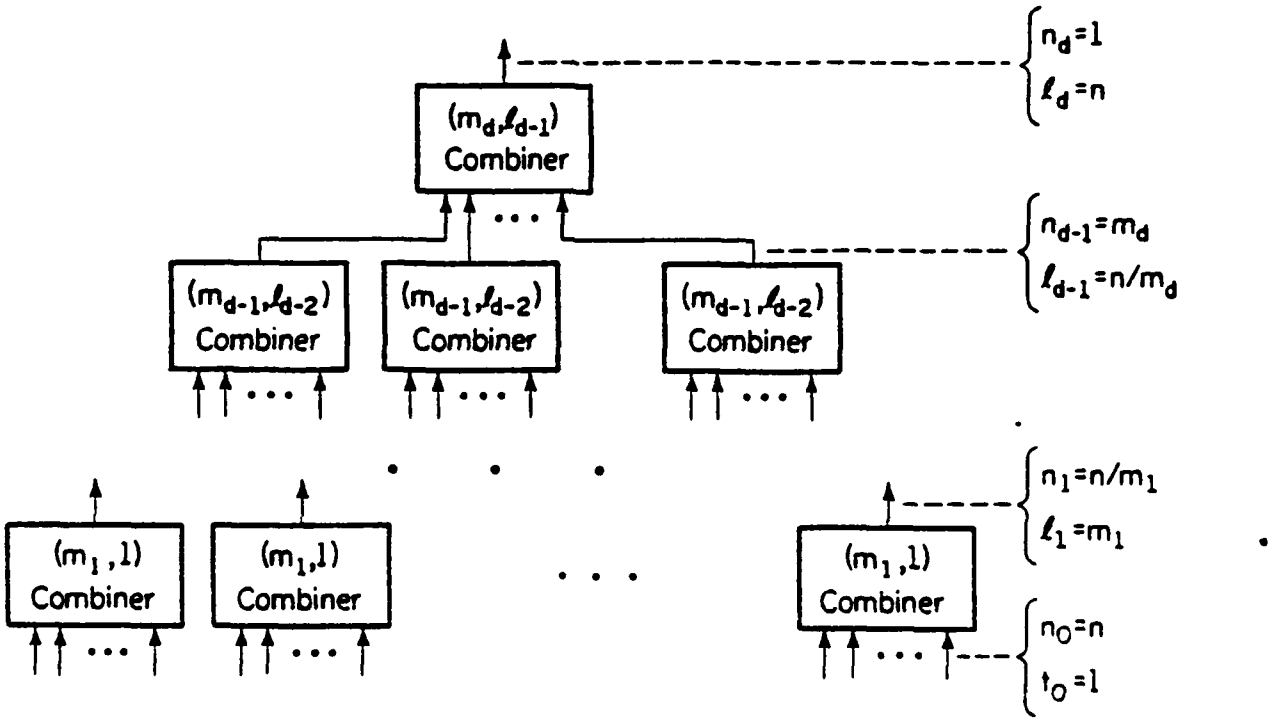


Figure 5.1. Diagram of combine-sort scheme.

5.2.2 Merge-Sort

Merge-sort is a special case of combine-sort, obtained when $n = 2^d$, and $m_1 = m_2 = \ldots = m_d = 2$. Merge-sort can be further specialized by specifying how to perform the combination of two sequenes, better known as merging.

Two important algorithms called *bitonic* merge and *odd-even* merge have been proposed by Batcher [Ba68]. Originally formulated for a network of comparators, both agorithms are also amenable to efficient VLSI implementation.

We shall make systematic use of bitonic merge, and correspondingly of bitonic sort, which exhibit a high degree of symmetry in the pattern of data interaction. Therefore, these algorithms are compactly described below, with these conventions: A[0:n-1] is the input array; d is a binary parameter specifying either increasing (d=0) or decreasing order (d=1); *COMPEX(a,b;d)* is a primitive operation which rearranges two numbers a and b in increasing or decreasing order depending upon the value of d. The array $A[0{:}n\text{-}1]$ is sorted by a call *B-SORT*($A[0{:}n\text{-}1]$,0) of the following procedure (where n and b are powers of2):[1]

procedure B-SORT($A[i{:}i+b\text{-}1]$,d)

begin if b=2 **then** ($A[i]$,$A[i+1]$) ← COMPEX($A[i]$,$A[i+1]$;d);

else begin B-SORT($A[i{:}i + \frac{b}{2} - 1]$,0), B-SORT($A[i + \frac{b}{2}{:}i + b\text{-}1]$,1);

B-MERGE($A[i{:}i+b\text{-}1]$,d)

end

end

procedure B-MERGE($A[i{:}i+b\text{-}1]$,d)

begin if b=2 **then** ($A[i]$,$A[i+1]$) ← COMPEX($A[i]$,$A[i+1]$;d)

else begin for each $0 \leq j < b/2$ **pardo**

($A[i+j]$,$A[i + \frac{b}{2} - 1+j]$) ← COMPEX($A[i+j]$,$A[i + \frac{b}{2} - 1+j]$;d);

B-MERGE($A[i{:}i + \frac{b}{2} - 1]$,$d$), B-MERGE($A[i + \frac{b}{2}{:}i+b\text{-}1]$,$d$)

end

end

(1) In the following algol-like program commas are used to separate concurrent steps, and semicolons are used to separate steps to be sequentially executed.

Odd-even merge is also a very interesting algorithm, but we do not describe it here, since we will not make direct use of it in our constructions. However, in Section 5.2.4, we shall describe the multiway-shuffle-combination algorithm, from which odd-even merge can be obtained as a special case.

5.2.3 Merge-Enumeration Combination

A parallel algorithm for the (ml)-combiner which we call *merge-enumeration* has been introduced in [P78] and is based on the following ideas. The m input sequences $S_0 \ldots, S_{m-1}$ are pairwise merged to compute for each $i, j \in \{0,1,\ldots,m-1\}$, and each $h \in \{0,1,\ldots,l-1\}$, and the number $C_{i,j}(h)$ of elements of sequence S_j that are less than the h-th element of sequence S_i. $C_{i,j}(h)$ is readily obtained as the difference of the ranks of this element in the merge of S_i and S_j and in S_i in the output sequence of the combiner; thus, to complete the operation, we simply need to store each element in the position specified by its rank. The primitive operation of the scheme -- the merging of two sequences -- can be done, for example, by Batcher's bitonic merger.

It can be shown [P78] that a proper implementation of merge-enumeration combination runs in time $O(\log(ml))$, that is, in time logarithmic in the size of the output sequence.

A very interesting case of the algorithm is obtained when $l = 1$ so that each of the input sequences, $S_0, \ldots, S_{m-1}$ consists of just one element, and merging degenerates to comparison-exchange. Instead, in this case the combiner itself becomes a sorter, and - more specifically - the *Muller-Preparata sorter* orginally proposed in [MP75].

In [P78], the merge-enumeration combination has been introduced to construct sorting algorithms for the shared-memory machine, that run in $O(\log n)$ time and require for their execution the smallest possible number of processors. In fact, Preparata has shown that, by choosing for the combine-sort scheme the values $d = \log\log n/\log(1/(1-\alpha))$, and $m_{d-i} = n^{\alpha(1-\alpha)^i}$ with $0 < \alpha < 1$, the resulting sorting algorithms can be executed in time $O(\log n/\alpha)$ by $O(n^{1-\alpha})$ processors. The sorting scheme corresponding to a given α can be described as follows. The n-input sequence is split into n^{α} (m_1 in our terminology) sequences of $n^{(1-\alpha)}$ (l_{d-1} in our terminology) elements each. These sequences are

sorted recursively, and then combined by an (m_d, l_{d-1})-combiner. The recursion stops when sequences of length 1 are obtained. We can obtain the values for d and $m_1, \ldots, m_d$ by a simple analysis of the unfolded recursive process.

In Section 6.3 we shall explore new significant choices of d and $m_1, m_2, \ldots, m_d$ that minimize the complexiy of a VLSI implementation of merge-enumeration combine-sort.

5.2.4 Multiway-Shuffle Combination

The *Multiway-shuffle* -combination algorithm has been introduced by Leighton [L84] (under the name of column sort), and is a generalization of the odd-even merge of Batcher [Ba68].

We recall that, if $N = N_1 N_2$, the N_1 -shuffle is defined as follows:

N_1-shuffle $(0,1,\ldots,N-1) =$

$$(0, N_2, \ldots, (N_1-1)N_2, 1, N_2+1, \ldots, (N_1-1)N_2+1, \ldots, N_2-1, 2N_2-1, \ldots, N-1).$$

The N_1-unshuffle is defined as the inverse permutation of the N_1-shuffle. It is easily seen that, on $N_1 N_2$ elements, the N_1-shuffle is identical to the N_2-unshuffle. A simple way to obtain the N_1 shuffle of a sequence of $N_1 N_2$ elements is to write the sequence into an $N_1 \times N_2$ array in row major order, and read the same array in column major order. (See Figure 5.2.)

We are now ready to describe the multiway-shuffle combination, which is also illustrated by a block diagram in Figure 5.3. $S_0, S_1, \ldots, S_{m-1}$ are the m sorted sequences of l elements each

$$S_i = (s_i(0), s_i(1), \ldots, s_i(l-1)), \quad i = 0, \ldots, m-1 \qquad 5.1$$

and they have to be combined in the sorted sequence

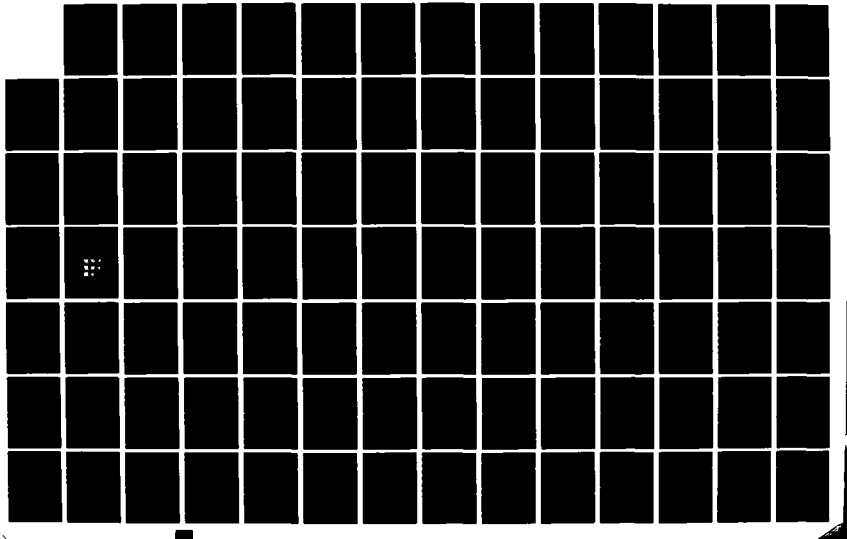
$$S = (s(0), s(1), \ldots, s(ml-1)) \triangleq \textit{combination}\ (S_0, \ldots, S_{m-1}). \qquad 5.2$$

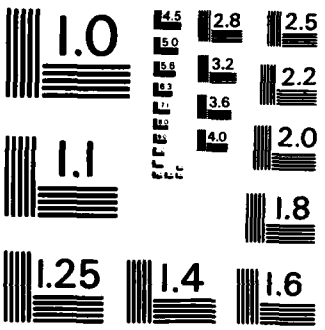
The algorithm consists of the following stages.

1. Apply a p-unshuffle to the sequence of ml elements obtained by concatenating $S_0, S_1, \ldots, S_{m-1}$. If we define the subsequences

AD-A161 562 THE AREA-TIME COMPLEXITY OF SORTING(U) ILLINOIS UNIV AT 2/3
URBANA APPLIED COMPUTATION THEORY GROUP G BILARDI
DEC 84 ACT-52 N00014-84-C-0149

UNCLASSIFIED F/G 12/1 NL





sequence : (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)

write in row-major

$$\begin{vmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{vmatrix}$$

read in column-major

3-shuffle : (0, 4, 8, 1, 5, 9, 2, 6, 10, 3, 7, 11)

Figure 5.2. Array definition of the multiway shuffle ($N_1 = 3, N_2 = 4$).

$$S_{ia} = (s_i(a), s_i(a+p), \ldots, s_i(l-(p-a))) \quad a = 0, \ldots, p-1 \qquad 5.3$$

it is easy to see that the output of the p-unshuffle is the concatentation of $S_{00}, S_{10}, \ldots, S_{m-1,0}, S_{01}, S_{11}, \ldots, S_{m-1,1}, S_{0,p-1}, S_{1,p-1}, \ldots, S_{m-1,p-1}$.

2. For $a = 0,1,\ldots,p-1$, recursively combine the m sequences $S_{0a}, S_{1a}, \ldots, S_{m-1,a}$ to obtain a sorted sequence U_a.

3. Apply a p-unshuffle to the concatenation of $U_0, U_1, \ldots, U_{p-1}$ and call the result U.

4. For a given even integer $w \geq 2(m-1)(p-1)$, which divides ml, split U into ml/w "windows" of w consecutive elements and sort each window. Call the resulting sequence U'.

5. Split sequence U' except the first $w/2$ and the last $w/2$ elements, into $ml/w-1$ windows of w elements, and sort each window. After this operation, we obtain S, the result of the combination.

The basic property of the multiway-shuffle combination that justifies the correctness of the algorithm, is the following.

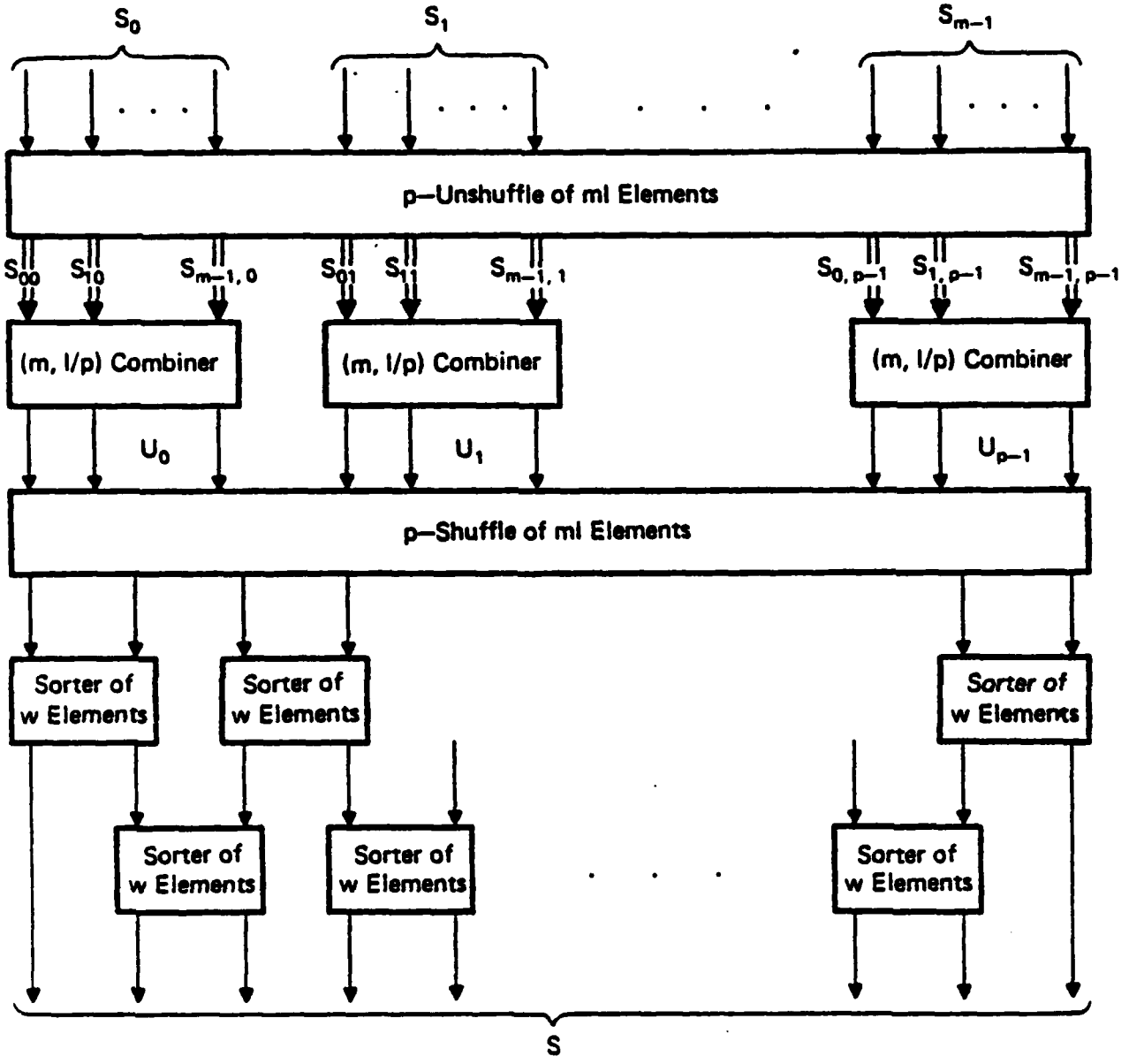


Figure 5.3. Block diagram of multiway-shuffle combiner. Single lines carry one element, double lines carry a sequence of $l/(mp)$ elements.

Property. If a given input element has position h_U in sequence U and position h_S in sequence S, then

$$|h_U - h_S| \leq (m-1)(p-1) . \qquad 5.4$$

More specifically, if $h_U = bp + a$, for some $b = 0,1,\ldots,ml/p-1$, then

$$h_L + (m-1)a - (m-1)(p-1) \leq h_S \leq h_L + (m-1)a \ . \qquad 5.5$$

Remark. As we have aready mentioned, the preceding algorithm is essentially equivalent to the one proposed in [L84]. The proof of the above property is also similar to the one proposed by Leighton, and it is therefore omitted here. However, our description of the algorithm is rather different from the one given in [L84], because we do not restrict ourselves to the case $p=m$, a case in which the unshuffling and the shuffling operations can be described as row-major to column-major transpositions of a suitable $l \times m$ array where the input sequences are orginally placed.

Remark. Odd-even merge is the special case of multiway-shuffle combination obtained for $m = p = 2$.

A simple analysis shows that the running time of multiway-shuffle (m,l)-combination when $p = m$, is $T = O((\log l / \log m) T_{sort}(m^2))$, where T_{sort} is the running time of the sorting algorithm that we chose to deploy in stages 4 and 5.

A combine-sort scheme based on multiway-suffle combination, with $m_1 = m_2 = \cdots = m_{d_0} = m$ (a constant independent of n), and recursively using multiway-shuffle to sort the windows in steps 4 and 5 results in a running time $T = O(\log^2 n)$. The net result is a rather cumbersome method to obtain the same performance as can be achieved by a simple merge-sort.

Nevertheless, multiway-shuffle combination is a remarkable algorithm, and it turns out to be very useful in some VLSI designs.

5.2.5 The AKS Network

Ajtai, Komlos, and Szemeredi [AKS83] recently proposed a sorting network (referred to hereafter as the *AKS network*), of $O(n \log n)$ comparators and $O(\log n)$ depth. Their construction is of great theoretical interest, for it shows that $O(n \log n)$ comparisons suffice to sort n elements, even under the constraint that comparisons be nonadaptively executed in $O(\log n)$, parallel stages. At present, the AKS network appears not suitable for practical implementation, due to the large value of the constants; however, improvements are conceivable that could make the network more attractive for real-world

applications.

The full description of the AKS network is too complex to be reported here.

5.2.6 Summary

The notion of (m,l)-combination provides a common framework to classify several known algorithms for parallel sorting.

In a trivial sense, every sorting algorithm falls in the combine-sort scheme, since an $(m,1)$-combiner is, by definition, a sorter of m elements. Indeed, both the Muller-Preparata and the AKS algorithms can be viewed as combination schemes only in this trivial sense. In fact, there is no intermediate stage of these algorithms at which the input multiset is partitioned in sorted blocks.

However, we have also seen non-trivial examples of combination-sort including bitonic and odd-even merge-sort, merge-enumeration combination, and multiway-shuffle combination.

In the algorithms we have just cited the same method is used to perform the combinations at all stages of the combine-sort scheme. However, different methods can be used at different stages, obtaining algorithms that we could generally call "hybrid combine-sort". Hybrid algorithms are indeed useful in VLSI applications, as we shall see in forthcoming sections.

We conclude this section with a graphic summary given in Figure 5.4.

5.3 PARALLEL ARCHITECTURES

A parallel algorithm is executed by a parallel architecture, which is a set of processors connected by data paths. When focussing on the interconnection pattern, the architecture can be formally viewed as a graph whose vertices correspond to processors, and whose edges correspond to data paths. Informally, we shall often refer to such graphs as networks, or computation graphs.

In the design of a VLSI system for the solution of a given computational problem, both the algorithm and the architecture can be chosen to minimize the area-time complexity.

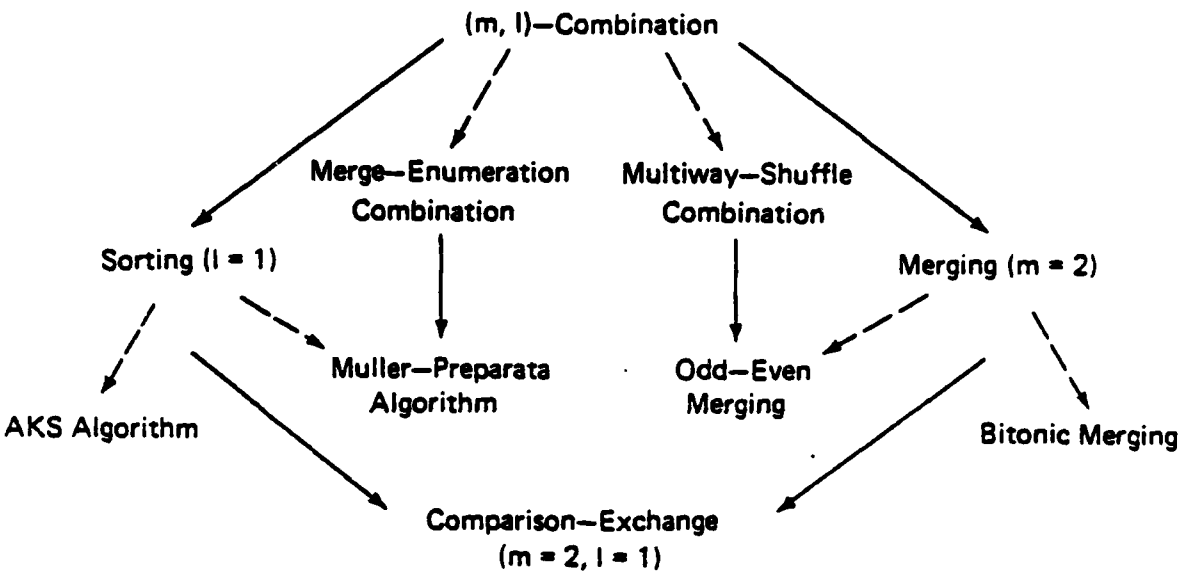


Figure 5.4. Hierarchy of fundamental operations in parallel sorting. A solid arrow points toward a subcase obtained by specializing the parameters defining the size of the input operands. A dashed arrow points toward a subcase obtained by specifying the algorithm by which the operation is performed.

Since the solutions of different problems requires different algorithms, it is a priori quite plausible that they also require different architectures. However, the experience gained by recent research in the field of VLSI computation (a representative, but by no means exhaustive, list of reference is: [BK81], [BK82], [BP84a], [BP84b], [BS84], [GKT79], [Ku82], [L81a], [L83], [Ls80a], [Me83], [MP84], [NMB83], [PV80], [PV81a], [PV81b], [T80], [T83a], [T83b]) shows that in several cases algorithms for different problems can be efficiently executed by the same achitecture. (For example the radix-2 Fourier transform and bitonic merging are both efficiently executed by the shuffle-exchange network.)

A detailed analysis of these cases reveals that although the nature of the operations performed on the input data may be radically different, the pattern according to which the processors exchange data among each other is exactly the same.

Preparata [P84] proposes to classify algorithms according to paradigms, determined exclusively by their communication patterns, and to characterize the architectures in relation to the paradigms that they can execute efficiently.

When a paradigm encompasses algorithms for several useful problems, the supporting architecture can be considered of the *broad-purpose* type, its capabilities being intermediate between those of a special-purpose achitecture, exclusively dedicated to a given task, and those of a general-purpose architecture that can execute any conceivable algorithms.

As observed in [P84], the results emerging from current research on the design of efficient VLSI systems for the solution of fundamental computational problems strongly suggest that a few powerful and highly regular architectures can be used to satisfy a majority of computational requirements.

The study of the sorting problem confirms this indication. As we shall see in the next chapters, all the known basic broad-purpose architectures, alone or combined in novel ways, are instrumental to obtain VLSI sorters with optimal area-time performance. For this reason, we briefly review them in the remainder of this section.

5.3.1 The Binary Cube and its Emulators

The binary cube. The *v -dimensional binary cube* [Pe77] is a network of $N = 2^v$ processors labelled from 0 to N-1 as $P(0), P(1), \ldots, P(N-1)$, with a direct connection (called a *link*) between each pair of processors whose binary numberings differ in exactly one position. If we let $C_j(h)$ be the integer obtained by complementing the coefficient of 2^j in the binary representation of integer h, then the j-th *dimension* of the cube is the set of edges $E_j \triangleq \{(h, C_j(h)) : 0 \leqslant h < N\}$. (See Figure 5.5)

Among the algorithms supported by the binary cube are those whose input is an array of data $A[0], A[1], \ldots, A[N-1]$ with component $A[i]$ intially loaded in processors $P(i)$, and whose execution consists of a sequence of steps such that at a given step, only the edges of one dimension are active. A pair of processors connected by an edge of that dimension exchange their data and operate on them.

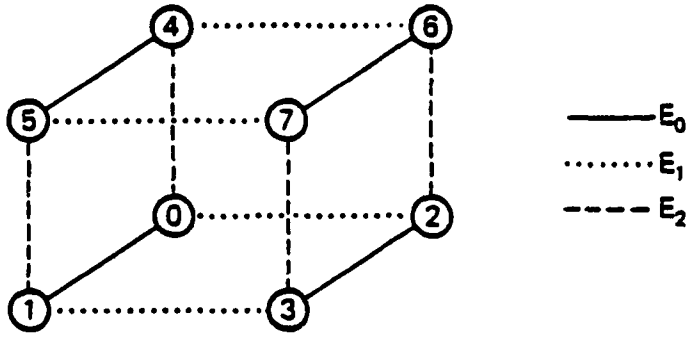


Figure 5.5. The binary cube and its dimensions. ($N=8, \nu=3$).

Such an algorithm belongs to a paradigm that can be easily described by giving the sequence of the dimensions in the same order as they have to be activated. With this convention, we can define two important paradigms

$$Ascend : (E_0, E_1, \ldots, E_{\nu-1}) \qquad 5.6$$

$$Descend : (E_{\nu-1}, E_{\nu-2}, \ldots, E_0). \qquad 5.7$$

These paradigms have been introduced in [PV81a] and [PV81b], where the reader can also find an extensive list of problems and algorithms complying with Ascend and Descend or simple variants there of. The recursive structure of Ascend and Descend algorithms is also elucidated in those papers.

If the operation executed at each step takes a constant amount of time, both the Ascend and the Descend algorithms are executed by the binary cube in $O(\nu) = O(\log N)$ time.

Although the cube is a theoretically fundamental network, it is not very attractive for practical implementations, because the number of edges per processor inceases with N. This drawback of the cube has naturally suggested the search for simpler networks capable of emulating the cube without significant loss in performance, at least in the execution of algorithms that can be cast in the Ascend and Descend paradigms. We describe now some of these emulators.

The shuffle-exchange network. For an even integer N, the *shuffle permutation* is the bijective function shuffle $(h) = 2h \bmod (N-1)$, for $h \in \{0, 1, \ldots, N-2\}$, and shuffle $(N-1) = N-1$. The *shuffle-exchange* is a graph with N vertices labelled from 0 to $N-1$, and with two kinds of edges: the *exchange-edges*, which are bidirectional, and connect vertices $2h$ and $2h+1$, for $h=0,1, \ldots, N/2-1$, and the *transfer edges*, which are directed, and go from vertex h to vertex shuffle(h), for $h = 0,1, \ldots, N-1$. (See Figure 5.6.)

As a network of processors $P(0), P(1), \ldots, P(N-1)$, the *shuffle-exchange* has several attractive features [St71] most of which are summarized by the fact that it can emulate, in a simple and elegant way, the Descend paradigm of the binary cube.

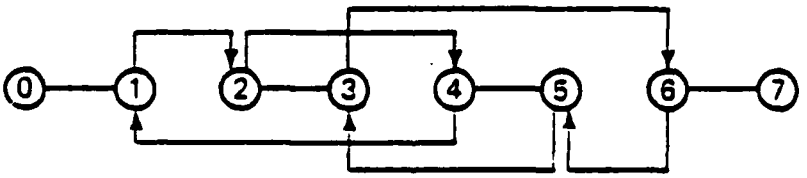


Figure 5.6. The shuffle-exchange graph for $N = 8$.

A Descend algorithm is in fact executed by the shuffle-exchange in ν phases, each of which consists of a *transfer step*, in which processor h sends its content to processor shuffle(h) along the transfer edge (h, suffle (h)) and an *operation step*, in which processors connected by an exchange edge communicate with each other and perform operations. An interesting property of the shuffle permutation, when N is a power of two, is that the binary spelling of shuffle(h) is the left cyclic shift of the spelling of h. Therefore, if $N = 2^{\nu}$, shuffle $(C_{\nu-j}(h)) = C_c$ (shuffle (h)), which means that h and $C_{\nu-j}$ will reside in processors connected by an exchange edge after j executions of the routing step, as required by the Descend paradigm on a ν-dimensional cube. This proves that the shuffle-exchange emulates the cube correctly. Moreover, since the ν-th power of the shuffle is the indentity, after ν steps all the items are back in the orginal processors (although they have been transformed by computation).

The inverse permutation of the shuffle, known as *unshuffle*, is also interesting. The unshuffle-exchange would in fact emulate the Ascend paradigm in the same way as the shuffle-exchange emulates the Descend paradigm. Indeed, the transfer edges of the shuffle-exchange are often defined as bidirectional so that both of shuffling and the unshuffling of the data can be accomplished in one transfer step.

In this case, it is easily seen that both the Ascend and the Descend algorithms have an $O(logN)$ running time on the shuffle-exchange network.

As to the area performance, the shuffle-exchange graph can be laid out in $A = \theta((N/logN)^2)$ area, which is optimal [KLLM83].

An attractive feature of the shuffle-exchange network is the simplicity of the emulation algorithm consisting of an alternation of transfer steps with operation steps, the transfer steps being all performed according to the same permutation, i. e. the shuffle. A natural question is whether we can obtain other emulators of the descend paradigm by using permutations different from the shuffle. This question is answered in [BJ84] where it is shown such permutations exist, but they are so closely related to the shuffle that there is nothing to loose in restricting our attention to the shuffle itself.

However, there are other interesting emulators of the binary cube, which use schemes to transfer the data more complex than a simple permutation.

The linear array. The linear array is a network of N processors $P(0), P(1), \ldots, P(N-1)$, with a bidirectional edge between $P(i-1)$ and $P(i)$, for $i = 1, 2, \ldots, N-1$.

The data contained in a linear array can be easiy shuffled (the content of $P(h)$ is sent to $P(\text{shuffle}(h))$, or unshuffled in $N/2$-1 transfer steps (for N even) as shown by [PV81a].

If $N = 2^{\nu}$, an operation step requiring the use of cube dimension $E_j (0 \leq j < \nu)$ can be peformed by the linear array as follows. (Refer to to Figure 5.7.) The entire array is decomposed in $N/2^{j+1}$ subarrays each of 2^{j+1} consecutive processors. Each subarray, in parallel and independently of the others, will shuffle its data to create the correct adjacencies required by the cube dimension E_j, and to allow for the execution of the operation step. Then, the original order of the data is restored by unshuffling the subarrays. This process requires one operation step, and $2\,(2^{j+1}/2-1) = 2^{j+1}-2$ transfer steps.

Initial	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Shuffle	0	1	2	3 ↔	4	5	6	7	8	9	10	11 ↔	12	13	14	15
of	0	1	2 ↔	4	3 ↔	5	6	7	8	9	10 ↔	12	11 ↔	13	14	15
Subarrays	0	1 ↔	4	2 ↔	5	3 ↔	6	7	8	9 ↔	12	10 ↔	13	11 ↔	14	15
Operation	0 —	4	1 —	5	2 —	6	3 —	7	8 —	12	9 —	13	10 —	14	11 —	15
Unshuffle	0′	4′ ↔	1′	5′ —	2′	6′ ↔	3′	7′	8′	12′ ↔	9′	13 ↔	10′	14 ↔	11′	15′
of	0′	1′	4′ ↔	2′	5′ ↔	3′	6′	7′	8′	9′	12′ ↔	10′	13′ ↔	11′	14′	15′
Subarrays	0′	1′	2′	4′ ↔	3′	5′	6′	7′	8′	9′	10′	12′ ↔	11′	13′	14′	15′
Final	0′	1′	2′	3′	4′	5′	6′	7′	8′	9′	10′	11′	12′	13′	14′	15′

Figure 5.7. Execution of operations of cube dimension E_2 with a linear array of N = 16 processors.

Thus, the Ascend and Descend paradigms can be executed by the linear array in ν operation steps and $\sum_{j=0}^{\nu-1}(2^{j+1}-2)<2N$ transfer steps. The same result holds for any cube paradigm in which all the ν dimemsions are activated exactly once, in any arbitrary order. In conclusion we obtain $T = O(N)$, and clearly $A = (N)$.

The rectangular mesh. The $(s \times t)$ rectangular mesh is a two-dimensional array of $N = st$ processors $P_{i,j}$, $i = 0, \ldots, s\text{-}1$, and $j = 0, 1, \ldots, t\text{-}1$, where each row and each column is interconnected as a linear array.

Let $N = 2^{\nu}$, $s = 2^{\sigma}$, and $t = 2^{\tau}$ and let the mesh be loaded with the input vector $A[0], \ldots, A[N-1]$ of the Ascend paradigm in column-major order, so that processor $P_{i,j}$ stores $A[i+sj]$ (refer to Figure 5.8).

If the processors were connected as a cube, with cube processors $P(i+js)$ corresponding to mesh processors $P_{i,j}$, dimensions $E_0, E_1, \ldots, E_{\sigma-1}$ would remain associated with the columns, and dimensions $E_{\sigma}, E_{\sigma+1}, \ldots, E_{\sigma+\tau-1}(\sigma+\tau = \nu)$ would remain associated with the rows of the mesh (see Figure 5.8).

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
3	7	11	15	19	23	27	31

Figure 5.8. A (4×8) rectangular mesh. When the indices of the cube processors are mapped into the array in column-major order dimensions E_0 and E_1 are associated with columns, and dimensions E_2, E_3, and E_4 are associated with rows. ($N = 32$, $s = 4$, $t = 8$, $\sigma = 2$, $\tau = 3$.)

The operation step associated to a given dimension can be then executed by the same technique used for linear arrays. If $0 \leqslant j \leqslant \sigma-1$, E_j is executed by suitable subarrays of the columns in $2^{j+1}-2$ transfer steps. If $\sigma \leqslant j \leqslant \sigma + \tau - 1$, E_j is executed by suitable subarrays of the rows in $2^{j-\sigma+1}$ transfer steps.

A simple analysis shows that using each deminsion once takes a global number of transfer steps near-square mesh ($s = t = \sqrt{N}$ if ν is even, or $s = 2t = \sqrt{N/2}$ if ν is odd) which works in $O(\sqrt{N})$ steps.

This result obviously applies to Ascend and Descend. It is useful to observe that the execution of an Ascend algorithm on the rectangular mesh can be viewed as the execution of an Ascend algorithm on the columns followed by an Ascend algorithm on the rows. Similarly, a Descend algorithm on the mesh consists of a Descend on the rows followed by a Descend on the columns.

Summarizing, a near-square mesh of N processors can execute algorithms in the Ascend and Descend paradigms in time $T = O(\sqrt{N})$. The layout area is clearly $A = O(N)$.

The Cube-Connected-Cycles. Referring to Figure 5.9, an $(s \times t)$-Cube-Connected-Cycle (CCC), with $s = 2^\sigma, t = 2^\tau, s \geqslant \tau$ is a network of $N = s\,t = 2^\nu$ modules and can be conveniently thought of as an $s \times t$ array of processors P_i^j $(0 \leqslant i < s, 0 \leqslant j < t)$ arranged as a martrix where j grows from left to right (as usual), whereas i grows from bottom to top. (Figure 5.9 illustrates a 4×8 CCC.) The CCC-processor P_i^j has *number* $h = j2^\sigma + i$ and corresponds to the cube processor $P(h)$. It follows that in the CCC the original cube indices are arranged in *column major order*. The columns of the $s \times t$ array are connected as cycles, with an edge between P_i^j and $P_i^{(j+1) mod s}$. The first τ rows $(0 \leqslant i < \tau)$ are associaed wih the τ highest dimensions of the cube; specifically row i contains an edge between each pair of processors who number differ exactly in bit position $\nu - \tau + i$. The dimensions of the cube are then divided into two groups: the *cycle dimensions* $E_0, \ldots, E_{\sigma-1}$ which pertain to interactions between pairs of elements in the same cycle and the *lateral dimensions* which pertain to interactions between pairs of elements of the same row.

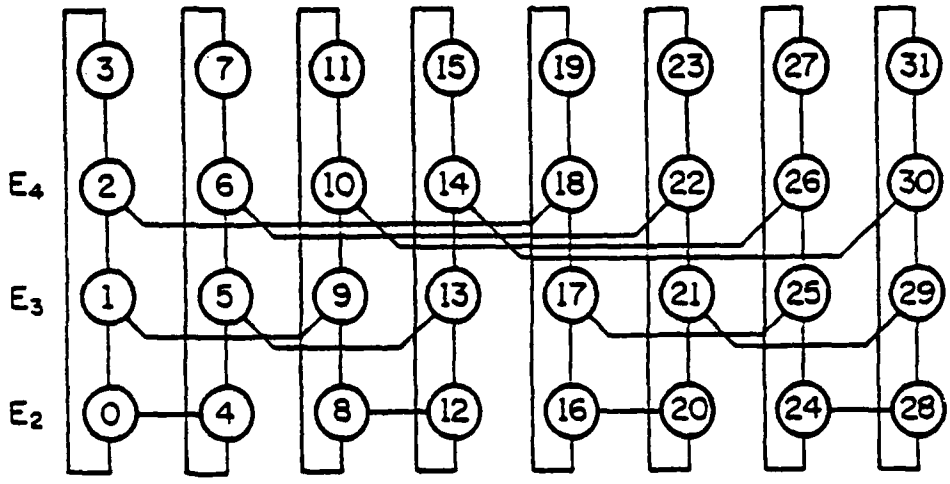


Figure 5.9. A 4×8 CCC. Processors are labelled with their numbers. The correspondence between rows and dimensions is also shown. ($\nu = 5$, $\tau = 3$).

To execute a cycle dimension E_j, the CCC works essentially as a set of $N/2^{j+1}$ independent linear arrays of length 2^{j+1} each of which performs the shuffle-operation-unshuffle procedure already described. Thus, the execution of $E_0, E_1, \ldots, E_{\sigma-1}$ globally require $O(s)$ time.

To execute a lateral dimension $E_{\sigma+j}$ $(0 \leq j < \tau)$, a sequence of s operation steps is peformed by the lateral connections in row α, interleaved with s cyclic shifts of the cycles. When the lateral dimensions have to be executed in the order $E_{\sigma}, \ldots, E_{\nu-1}$, their execution can be overlapped by a suitable pipelining technique, so that the total time is just $O(s)$.

In conclusion a $(s \times t)$-CCC can execute an Ascend algorithm in $T = O(s)$ computation time. A similar performance can also be achieved for a Descend algorithm.

As to the area, it can be shown [PV81] that a $(s \times t)$-CCC with $\tau \leq s$ and $t \leq \sqrt{N}$ can be laid out in $A = O(t^2)$ area which is optimal.

For a given $N = 2^p$, by chosing s in the range $[\log N, \sqrt{N}]$ we can achieve a perfomance $AT^2 = O(t^2 s^2) = O(N^2)$ for any computation time $T \in [\Omega(\log N), O(\sqrt{N})]$.

Comparison of Cube Emulators. The performance of the emulators of the cube we have described for the Ascend and the Descend paradigm is summerized in Table 5.1. The shuffle-exchange, the linear array and the mesh all have the same $AT^2 = \theta(N^2)$ performance, which is optimal. In the following chapters we usually deploy cube-connected-cycles for the execution of Ascend and Descend algorithms, especially because of its area-time trade-off feature that allows to choose the value of the computation time from a wide range. It must also be said that, for $(T = O\sqrt{N})$, the mesh is usually preferable for the simplicity of its interconnection. For $T = O(\log N)$, the shuffle-exchange is attractive for the elegance of the emulation algorithm. However, the optimal layout of the shuffle-exchange is very irregular, which is not a desirable feature for VLSI systems.

The linear array is indeed a poor emulator of the cube, at least when judged by its area-time performance. However, it is very useful as a component of more complex networks, as we have already seen in the case of the rectangular mesh and of the CCC.

TABLE 5.1. AREA-TIME PERFORMANCE OF CUBE EMULATORS

Performance / Architecture	TIME	AREA
Suffle-Exchange	$T = \Theta(\log N)$	$A = \Theta(N^2/T^2)$
Linear Array	$T = \Theta(N)$	$A = \Theta(N^3/T^2)$
Square Mesh	$T = \Theta(\sqrt{N})$	$A = \Theta(N^2/T^2)$
CCC	$T \in [\Omega(\log N)), O(\sqrt{N})]$	$A = \Theta(N^2/T^2)$

5.3.2 The Tree and the Orthogonal Trees

The binary tree. Several computations require the N-fold replication of a given data item, or some kind of combination of N distinct data items to generate a single one. These operations are efficiently executed in $\theta(\log N)$ steps by a fully balanced *binary tree* with $N = 2^v$ leaves and N-1 internal nodes. This graph can be laid out in $\theta(N)$ area if there is no constraint on the placement of the leaves, and in $\theta(N \log N)$ area the leaves must be placed on the boundary of the layout region [BK80].

The orthogonal trees. (Refer to Figure 5.10) The two-dimensional *orthogonal tree network* (OT) [L81, \\1B83] consists of $N = n^2$ processors $P_{i,j}$ $(i,j = 0,1, \ldots, n\text{-}1)$, and $2n$ fully balanced binary trees $CT_0, \ldots, CT_{n-1}$ (the column trees), and $RT_0, \ldots, RT_{n-1}$ (the row trees). The leaves of CT_j are then processors $P_{0,j}, \ldots, P_{n-1,j}$ and the leaves of RT_i are the processors $P_{i,0}, \ldots, P_{i,n-1}$.

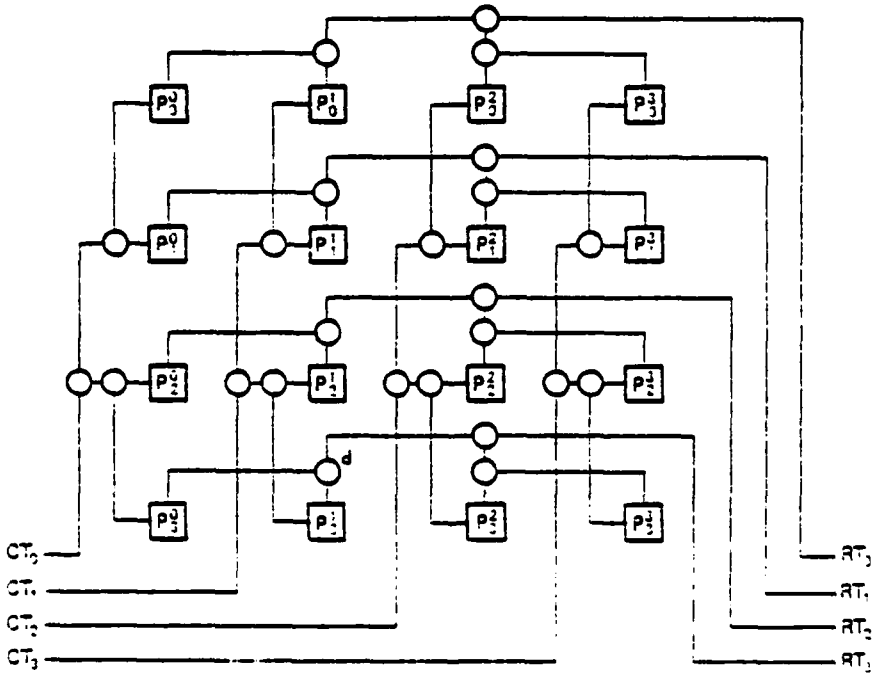


Figure 5.10. Orthogonal-tree network for $N = 16$.

Optimal layouts of the OTs have area $A = \theta(N \log^2 N)$ [L81]. Algorithms consisting of a constant number of replication and combination operations along the row and the column trees, are executed by the OTs in time $T = O(\log N)$.

The OT network is very versatile and will be used is several of our sorters. Multidimensional OTs can also be defined. An interesting application of three-dimensional OTs to matrix multiplication can be found in [PV80].

CHAPTER 6

OPTIMAL VLSI SORTERS FOR KEYS OF LENGTH $k = \log n + \theta(\log n)$

6.1 INTRODUCTION

In most of the investigations on VLSI sorting the length of the keys has been assumed to be of the form $k=(1+\alpha)\log n$, for some constant $\alpha > 0$. Since the results of those investigations are indeed valid as long as $(1+\alpha_1)\log n \leq k \leq (1+\alpha_2)\log n$, for some constants $\alpha_2 > \alpha_1 > 0$, it is slightly more appropriate to refer to a length of the form $k=\log n + \theta(\log n)$, not to suggest that, say, $k=2\log n + \log\log n$ is excluded by our considerations.

In retrospect, we can jusify the attention given to the case $k=(1+\alpha)\log n$ for two reasons: (1)this case of the sorting problem is the easiest (or the least difficult) to analyze, and (2) its complete solution is instrumental to make progress on other cases. While the second reason will be substantiated only in Chapter 7, where we will show that a sorter of key with $k=(1+\alpha)\log n$ is a useful building block or sorters of both short and long keys, the first reason can be already explained on the basis of the lower bound results of Part I.

In fact, while short and long keys have to be studied with the more sophisticated square-tessellation technique, the case $k=(1+\alpha)\log n$ - which partially overlaps with medium-length keys ($\alpha < 1$), and partially with long keys ($\alpha \geq 1$) - can be analized by the bipartition technique (although, as we have observed in Section 4.2, the dependence of the complexity on α, for $\alpha > 1$, can be really understood only by the square tessellation bound).

Indeed, from Theorems 4.14, 4.15, 4.16, and the assumption $k > (1+\alpha_1)\log n$ for some $\alpha_1, > 0$, we obtain

$$AT^2 = \Omega(n^2 \log^2 n) \quad 6.1$$

$$T = \Omega(\log n) \quad 6.2$$

$$A = \Omega(n \log n) \quad 6.3$$

In this chapter, we will study several VLSI sorters, the analysis of which will allow the conclusion that the optimality curve of the $(n, \log n + \theta(\log n))$-sorting problem is described by

$$A = \theta(n^2 \log^2 n / T^2), \quad T \in [\Omega(\log n), O(\sqrt{n \log n})] \quad 6.4$$

As we shall see, the main difficulty in obtaining optimal VLSI sorters for $k = \log n + \theta(\log n)$ consists in designing the appropriate architecture. For the algorithms instead, it will be sufficient to resort to (minor adaptations of) the known results reviewed in Section 5.2. The situation will be different for short and long keys, which will require new algorithms as well as new architectures.

This fact is not without explanation. The first parallel sorting algorithms have been conceived for either the shared-memory-machine or the network-of-comparators models of computation, whose primitive operations are at the word level. Thus, the keys were treated as indivisible entities that maintain their identity throughout the entire computation.

The indivisibility of the key is a very restrictive constraint in the VLSI model of computation, and conflict with area-time optimality.

Indeed, to sort short keys it is not convenient to maintain a list encoding of the input multiset in the intermediate stages of the computation, because it is very inefficient, and requires superfluous bandwidth in transmission. Thus, no algorithm that maintains the identity of the keys can achieve optimality.

The same conclusion is true also for rather long keys ($\log n = o(k)$) but for a different reason. The list representation is indeed efficient in this case, but we still need to fragment the keys to avoid a large primary flow. In fact, we have already seen that even when the "indivisibility" of keys is required only at the I/O ports (word-locality) the AT^2 complexity of (n,k)-sorting is asymptotically quadratic in k (Theorem 4.14), while without this restriction the complexity is only linear in k (Theorem 4.18).

The case $k = \log n + \theta(\log n)$ is made special (and, superficially, simpler than others) by two circumstances. One is that the list encoding is optimal for this length (within a constant factor). The other is that any strategy to decrease the primary flow below $\theta(n \log n)$ by suitably decomposing the keys would fail to yield better area-time performance due to the presence of an irreducible $\theta(n \log n)$ secondary flow.

In conclusion, for $k = \log n + \theta(\log n)$ the indivisibility of the keys (word-locality) is not a drawback, and classical sorting algorithms turn out to be instrumental to obtain area-time optimal designs.

With this premise, we now turn our attention to the effective construction of VLSI sorters. Many designs have been proposed in the early literature, and reference [T83] surveys several of them. Here, we recall only the designs that come closer to $AT^2 = \Omega(n^2 \log^2 n)$ lower bounds, which are:

(i) a *mesh-connected* bitonic sorter ([Ba68], [T80], [TK77], [NS79]) with optimal performance $A = \theta(n^2 \log^2 n / T^2)$ at $T = \theta(\sqrt{n})$, and other four designs all with suboptimal performance $A = \theta(n^2 \log^4 n / T^2)$ which are:

(ii) a *shuffle-exchange* bitonic sorter at $T = \theta(\log^3 n)$ ([St71][T80][KLLM83])

(iii) a *cube-connected-cycles* bitonic sorter, for $T \in [\Omega(\log^3 n), O(\sqrt{n \log n})]$ ([PV81])

(iv) a pipelined *Batcher's network*, at $T = \theta(\log^2 n)$ (VLSI estimate in [T83])

(v) an *orthogonal-tree-connected* ([L81], [NMB83]) Muller-Preparata sorter [MP75] at $T = \theta(\log n)$.

The $\theta(\log^2 n)$ gap between lower and upper bound for AT^2 exhibited by the last four designs has indeed been one of the original motivations for the work reported in this thesis. The remainder of this chapter is devoted to the discussion of VLSI sorters with optimal performance. For several of them, a description is already available in the literature.

Section 6.2 is devoted to bitonic sorting. Our approach will consist in focussing on the underlying paradigm, which is of the cube type, but more complex than the Ascend or the Descend paradigms, and in constructing efficient architectures for that paradigm.

Section 6.3 is devoted to merge-enumeration sort, which allows to achieve minimum computation time $T = O(\log n)$. New architectures are also considered, based on a mixing of orthogonal trees and cube-connected-cycles.

Section 6.4 concludes the chapter with a brief report on other area-time optimal networks implementing multiway-shuffle sort and the AKS algorithm.

6.2 NETWORKS FOR BITONIC SORTING

6.2.1 The Bitonic Sorting Paradigm

The bitonic sorting of $n = 2^\nu$ elements (reviewed in Section 5.2.2) consists of ν merging phases, $M_0, M_1, \ldots, M_{\nu-1}$, with phase M_i performing the merging of pairs of sequences of length 2^i.

Bitonic merging, on the other hand, complies with the Descend paradigm of the binary cube [PV81a], so that the execution of phase M_i on the binary cube requires the successive use of dimensions $E_i, E_{i-1}, \ldots, E_1, E_0$. Thus, the schedule of use of the dimensions for a complete sorting is the one shown below (Figure 6.1), which will be called the *bitonic sorting paradigm*.

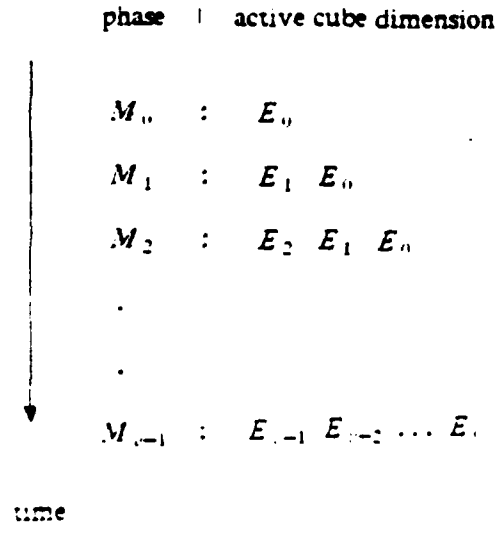


Figure 6.1. The bitonic sorting paradigm.

We shall now analyze the performance of some of the known emulators of the cube on the bitonic sorting paradigm, and we shall then propose new and more efficient emulators.

Some considerations will put the problem in the proper perspective, and will indicate the difficulties to be overcome for its solution.

The area-time performance of the emulators of the cube reported in Section 5.3.1 pertains to the Ascend and Descend paradigms. The area is estimated under the assumption that links between processors are realized with unit bandwidth, so that they can be laid out in unit width, and the computation time is estimated under the assumption that both operation and transfer steps take unit (or constant) time.

When the actual data processed by the algorithm have length k, the estimate on the AT^2 measure must be multiplied by k^2, although we can usually still choose whether the penalty is to be paid in area or in time. In fact, for a given b with $1 \leq b \leq k$, if we realize all the links with bandwidth b we obtain an area $A_b = b^2 A_1$, and - usually - a time $T_b = \frac{k}{b} T_1$, so that $A_b T_b^2 = k^2 A_1 T_1^2$.

The reason for which we cannot claim that T_b always equals $(k/b)T_1$ is that, although a larger bandwidth automatically yields a proportional speed-up in transfer steps, it does not guarantee a speed-up in operation steps. However, most of the time operation steps can be performed in k/b time, at least as long as k/b is not small. For example, in sorting, the operation step usually involves a comparison-exchange, which can be performed in time k/b (and area b^2) as long as $k/b = \Omega(\log k)$, or equivalently, $b = O(k/\log k)$.

Thus, the optimal emulators of the Ascend and Decend paradigms, achieve a performance $AT^2 = O(N^2 k^2)$ on operands of length k. If $N = n$, and $k = \log n + \theta(\log n)$, $AT^2 = O(n^2 \log^2 n)$.

In order to attain the $AT^2 = \Omega(n^2 \log^2 n)$ lower bound for $(n, \log n + \theta(\log n))$-sorting by means of the bitonic algorithm, we must be able to execute the entire bitonic sorting paradigm in the same order of time as the much simpler Descend paradigm. It is indeed surprising that this is possible.

6.2.2 The Linear Array

Although the linear array is far from being an optimal emulator of the cube even for the simple Ascend and Descend paradigms, the study of its performance on the execution of bitonic sorting will provide us with some useful insights.

With reference to the discussion of Section 5.3.1, we shall consider a linear array of $N = n$ processors, where n is the number of keys to be sorted. Each processor will be endowed with $O(k)$ bits of memory, to allow the storage of a key of length k, and with a serial comparator-exchange that works in $O(k)$ time. We can then lay out a processor in a region of $O(1)$ width and $O(k)$ height. The connection between $P(i\text{-}1)$ and $P(i)$ can be realized with bandwidth k, to allow the execution of transfer steps in one time unit. The entire array can then be easily laid out in $O(n) \times O(k)$ area. (See Figure 6.2.)

The running time of bitonic sorting on the linear array is readily estimated. The operation steps are $(\nu + 1)\nu/2 = O(\log^2 n)$ in number, and each takes $O(k)$ time, so that, globally, the comparison-exchange steps take $T_1 = O(k\log^2 n)$ time. As for the data transfer we recall from Section 5.3.1 that execution of E_j is done in $2^{j-1} - 2$ steps. Since the bitonic sorting uses dimension E_j exactly $\nu - j$ times, globally the transfer steps take $T_2 < \sum_{j=1}^{\nu-1} (\nu - j)2^{j-1} = O(n)$ time. In conclusion, for $k = \log n + \theta(\log n)$, T_1 is negligible with respect to T_2, and the total sorting time is $T = O(n)$.

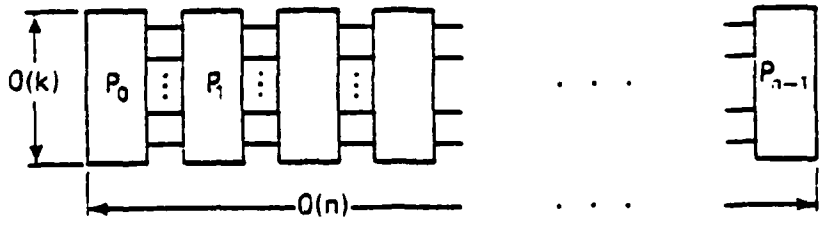


Figure 6.2. Layout of linear array for bitonic sorting.

At first, this is surprising since we would intuitively expect that the bitonic sorting paradigm, which consists of $\nu(\nu-1)/2$ steps on the cube, would require more time than the descend paradigm consisting of an ν steps on the cube. However, a closer analysis reveals that the dimensions more frequently used by bitonic sorting are the lowest, which also happen to be the ones that require less time on the linear array. Obviously, this is a fortunate accident, but the principle that the most frequently used dimension should be the ones with the fastest execution will be a useful guideline for subsequent developments.

6.2.3 The Mesh

Let $n = 2^\nu$ and, for simplicity, let ν be even. We consider now the execution of bitonic sorting with a $(\sqrt{n} \times \sqrt{n})$-mesh. The n processors of the mesh will be equippped with a serial comparator, and with $O(k)$ bits of storage. They will be laid out in an $O(b) \times O(b)$ region, with $\sqrt{k} \leq b \leq k$, to allow bandwidth b for both the horizontal and the vertical connections. The global layout area is then $A = O(b^2 n)$.

The $O(\log^2 n)$ comparison-exchange steps globally take $O(k \log^2 n)$ time, which, for $k = \log n + \theta(\log n)$, will be completely negligible with respect to the time used for transfer steps.

As we have seen in Section 5.3.1, the execution of dimensions E_j and $E_{\nu/2+j}$ uses $2^{j+1} - 2$ transfer steps, for $j = 0, 1, \ldots, \nu/2 - 1$. A simple calculation shows that the total number of transfer steps is $O(\sqrt{n} \log n)$, and hence the computation time is of order $O(\sqrt{n} \log^2 n / b)$.

Summarizing, if $k = \log n + \theta(\log n)$, $A = O(b^2 n)$ and $T = O(\sqrt{n} \log^2 n / b)$, so that $AT^2 = O(n^2 \log^4 n)$ which is within a factor $O(\log^2 n)$ of the lower bound. We observe that, since $b \in [\sqrt{k}, k]$, the computation time can be chosen in the range $T \in [\Omega(\sqrt{n} \log n), O(\sqrt{n} \log^{3/2} n)]$.

By using a more efficient implementation of bitonic sorting on the mesh, [TK77] and [NS79] have managed to reduce the number of transfer steps to $O(\sqrt{n})$. Using this result in the above analysis we obtain the following theorem.

Theorem 6.1. Bitonic sorting of n keys of length $k = \log n + \theta(\log n)$ can be executed by a

$(\sqrt{n} \times \sqrt{n})$-mesh with optimal $AT^2 = \theta(n^2 \log^2 n)$ for $T \in [\Omega(\sqrt{n}), O(\sqrt{n \log n})]$.

The original description of the algorithm in [TK77] and [NS79] is rather complex, but it can be simplified by using an approach that focusses on the paradigm.

Indeed, in the next section we shall develop a framework in which the optimal algorithm for bitonic sorting on the mesh can be easily obtained as a specialization of a general principle.

6.2.4 Efficient Use of Cube Emulators for Arbitrary Paradigms.

Any emulation procedure by which a given graph $G=(V,E)$, with $|V| = N = 2^\nu$, emulates the ν-dimensional binary cube is based on a one-to-one correspondence between the vertices of G and the vertices of the cube, such that $v \in V$ corresponds to $f(v) \in \{0,1,\ldots,N-1\}$. The emulation procedure is correct when, if the processor associated with $v \in V$ is initially loaded with the same input data $A[f(v)]$ that in the cube are loaded in processor $P_{f(v)}$, then upon termination the processor associated with v contains the same output data $A[f(v)]$ that the cube contains in $P_{f(v)}$.

We investigate now the possibility of modifying the function f for a *fixed* graph G. In particular, let $\sigma(0),\ldots,\sigma(\nu-1)$ be a permutation of the dimensions $(0,\ldots,\nu-1)$, and let $\pi(0),\ldots,\pi(N-1)$ a permutation of $\{0,\ldots,N-1\}$ such that if h has the binary representation $h_{\nu-1}h_{\nu-2}\ldots h_0$, then $\pi(h)$ has the representation $h_{\sigma^{-1}(\nu-1)}h_{\sigma^{-1}(\nu-2)}\ldots h_{\sigma^{-1}(0)}$. Thus, if h and h' are connected by an edge in E_j, then $\pi(h)$ and $\pi(h')$ are connected by an edge in $E_{\sigma(j)}$. We consider the correspondence between G and the cube defined by $f_\sigma(v) \triangleq \pi(f(v))$ (see Figure 6.3).

If for the pair (G,f) there is a procedure that emulates the execution of dimension E_j in time T_j, for the pair (G, f_σ) the same procedure will emulate the execution of dimension $E_{\sigma(j)}$ in time T_j.

Given a paradigm consisting of an arbitrary schedule of use of the cube dimensions $(E_{d_1}, E_{d_2}, \ldots, E_{d_M})$, we can ask for which the pair (G, f_σ) achieves the minimum emulation time.

The answer is not difficult. Let $\mu(j)$ be the number of times that E_j is used by the paradigm (μ is the multiplicity function of multiset $\{d_1, d_2, \ldots, d_M\}$). If $p_0, p_1, \ldots, p_{\nu-1}$ is the sequence of the

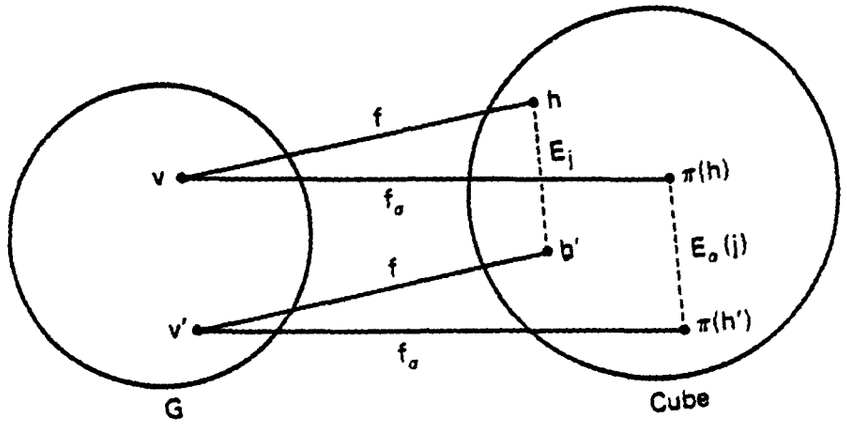


Figure 6.3. Correspondences f and f_σ between G and the cube.

dimensions arranged from the most to the least frequently used (i.e., $\mu(p_0) \geq \mu(p_1) \geq \cdots \geq \mu(p_{\nu-1})$), and $q_0 q_1 \cdots q_{\nu-1}$ is the sequence of dimensions arranged in order of nondecreasing execution time under correspondence f (i.e., $T_{q_0} \leq T_{q_1} \leq \cdots \leq T_{q_{\nu-1}}$), then it is relatively simple to realize that the minimum emulation time is achieved by the permutation

$$\sigma(q_h) = p_h, \qquad h = 0,1,\ldots,\nu-1 \tag{6.5}$$

and is given by

$$T_\sigma = \sum_{h=0}^{\nu-1} \mu(p_h) T_{q_h}. \tag{6.6}$$

Example (Bitonic Sorting on the Mesh). Let $G=(V,E)$ be the square mesh, and $V = \{(i,j): 0 \leq i,j < 2^{\nu/2}\}$. In Section 5.3.1 we have described an emulation procedure with respect to a column major numbering of the vertices, so that $f(i,j) = 2^{\nu/2} j + i$. The corresponding execution times are given by

$$T_q = T_{\nu/2+q} = 2^{q+1} - 2, \text{ for } q = 0,1,\ldots,\nu/2-1.$$

Thus,

$$T_0 = T_{\nu/2} < T_1 = T_{\nu/2+1} < \cdots < T_{\nu/2-1} = T_{\nu-1}$$

and $(q_0, q_1, \ldots, q_{\nu-1}) = (0, \nu/2, 1, \nu/2+1, \ldots, \nu/2-1, \nu-1)$.

For the bitonic sorting paradigm $(E_0; E_1, E_0; \ldots; E_{\nu-1}, \ldots, E_0)$ dimension E_q is used $\mu(q) = \nu - q$ times, so that $\mu(0) > \mu(1) > \cdots > \mu(\nu-1)$ and $(p_0, p_1, \ldots, p_{\nu-1}) = (0, 1, \ldots, \nu-1)$. From Eq. 6.6, the permutation σ that minimizes the emulation time is

$$\sigma = (\sigma(0), \ldots, \sigma(\nu-1)) = (0, \nu/2, 1, \nu/2+1, \ldots, \nu/2-1, \nu-1).$$

For the emulation time, Eq 6 yields

$$\begin{aligned} T_\sigma &= \sum_{h=0}^{\nu-1} \mu(p_h) T_{q_h} \\ &= \sum_{h=0}^{\nu/2-1} (\mu(p_{2h}) T_{q_{2h}} + \mu(p_{2h+1}) T_{q_{2h+1}}) \\ &= \sum_{h=0}^{\nu/2-1} (\mu(\nu-2h) T_h + \mu(\nu-2h-1) T_{\nu/2+h}) \\ &= \sum_{h=0}^{\nu/2-1} (2\nu-4h-1)(2^{h+1}-2) \\ &= O(2^{\nu/2}) = O(\sqrt{n}). \end{aligned}$$

The permutation σ and the numbering f_σ are illustrated in Figure 6.4, for $\nu = 4$. In general, if i and j respectively have binary representations $j_{\nu/2-1} j_{\nu/2-2} \cdots j_1 j_0$ and $i_{\nu/2-1} i_{\nu/2-2} \cdots i_1 i_0$, then $h = f(i,j) = 2^{\nu/2} j + i$ has the binary representation $j_{\nu/2-1} \cdots j_0 i_{\nu/2-1} \cdots i_0$, and $f_\sigma(i,j) = \pi(h)$ has the binary representation $j_{\nu/2-1} i_{\nu/2-1} \cdots j_1 i_1 j_0 i_0$. □

The I/O Format. Usually the format of the input array $A[0], \ldots, A[n-1]$ and of the output array $A[0], \ldots, A[N-1]$ are imposed a priori on our emulator G by global system considerations, and must be consistent with the I/O format of other parts of the system that are interfaced with G. As a consequence, at the end of the input phase, the data may be loaded into the processors of G in an order that differs from the one required by the emulation algorithm. A similar situation might occur for the out-

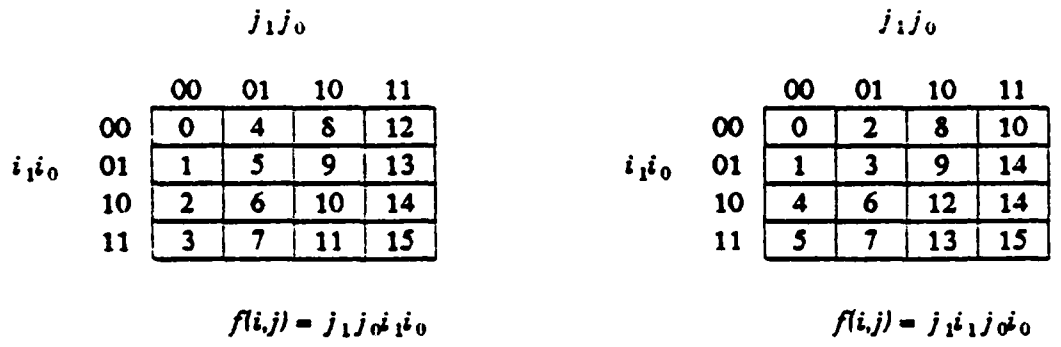


Figure 6.4. Column-major numbering f, and optimal numbering f_{opt} for the bitonic sorting paradigm on a (4x4) mesh.

put data.

In such a situation the emulation procedure must be preceded and followed by a suitable permutation of the data. This problem can be solved by resorting to the Benes permutation algorithm.

Originally formulated for a network of switches [Be64], Benes' algorithm can be also cast in a cube paradigm consisting of a Descend followed by an Ascend [PV81a]. The schedule of use of the dimensions is then

$$(E_{\nu-1}, E_{\nu-2}, \ldots, E_1, E_0, E_1, \ldots, E_{\nu-2}, E_{\nu-1}).$$

At each operation step, a pair of processors connected along the active dimension, may or may not exchange their data, according to the value of a control bit. Each permutation of size 2^ν can be realized by this algorithm, by a suitable choice of the control bits.

Although the parallel computation of the control bits for an arbitrary permutation is not a simple task, in the application we are considering the permutations of data to be realized are known at design time. Thus, the control bits can be precomputed and stored in the processors, provided that each processor is endowed with $O(\nu) = O(\log N)$ storage.

In conclusion, by adding to the computation time the usually negligible overhead corresponding to a constant number of Ascend and Decend algorithms, any (a priori known) I/O format can be combined with any correspondence of processors between the cube and the emulator.

The Benes permutation algorithm could be exploited to build emulation procedures more sophisticated than the one described in this section. In fact, for a paradigm in which the frequency of use of the dimensions is strongly time dependent, it may be convenient to dynamically change the allocation of the dimensions during the execution of the algorithm. This approach, however, will be not further pursued in this thesis (being inapplicable to the sorting problem).

6.2.5 The Cube-Connected-Cycles.

We have seen that the mesh-connected bitonic sorter is area-time optimal for slow computation. To obtain faster sorters we turn our attention to the CCC which we already know to be optimal for the Descend paradigm in a wide range of computation times.

As we have seen in Section 5.3.1, in an $(s \times t)$-CCC $(s = 2^{\sigma}, t = 2^{\tau}, s \geq \tau, n = st = 2^{\nu})$ the cube dimensions are naturally divided into two groups: the cycle dimensions $E_0, E_1, \ldots, E_{\sigma-1}$, and the lateral dimensions $E_{\sigma}, E_{\sigma+1}, \ldots, E_{\sigma+\tau-1}$ $(\sigma+\tau = \nu)$. A cycle dimension E_j $(0 \leq j \leq \sigma-1)$ is executed in $T_j = 2^{j+1}-2$ transfer steps. A lateral dimension $E_{\sigma+j}$ $(0 \leq j \leq \tau-1)$ is executed by pipelining the data around the complete cycle, and therefore uses uses $T_{\sigma+j} = s$ transfer steps. However, when all the lateral dimensions have to be executed consecutively, $O(s)$ transfer steps are sufficient (rather than $O(\tau s)$) because the pipelined mode of operation allows to overlap the execution of different dimensions.

In a paradigm like bitonic sorting, where in several merging phases only some of the lateral dimensions are executed, the emulation procedure becomes inefficient. In fact, each of the last $\tau = O(\log n)$ merging phases $M_{\sigma}, \ldots, M_{\sigma-1}$, requires the use of a set of cosecutive lateral dimensions (more specifically, $E_{\sigma-1}, \ldots, E_{\sigma}$ are used during phase $M_{\sigma+1}$), and therefore takes $O(s)$ transfer steps. Thus, the CCC executes the sorting paradigm in $O(\tau s) = O(s \log n)$ transfer steps.

We recall from the discussion of Section 6.2.1 that to attain the lower bound for sorting we need to keep the number of transfer steps in the bitonic sorting paradigm of the same order as in the Descend paradigm, which, for the CCC, is $O(s)$.

At first, the problem might seem similar to the one already encountered for the mesh, where we have reduced the number of transfer steps from $O(\sqrt{n}\log n)$ to $O(\sqrt{n})$ by a more appropriate allocation of the dimensions. However, for the CCC we face a more difficult situation because the dimensions already obey the principle that the most frequently used ones are those with the least execution time. The problem is that most of the dimensions have a high execution time.

Thus, a fast and efficient execution of the bitonic sorting paradigm requires the development of new networks. In the next two sections we shall describe the pleated-cube-connected-cycles, and the mesh of cube-connected-cycles, and show that they are area-time optimal emulators of the bitonic sorting paradigm.

6.2.6 The Pleated-Cube-Connected-Cycles (PCCC)

Description. There is a basic observation that, when recursively applied, leads to the modification of the CCC into the PCCC, and to a performance gain. The informal argument goes as follows: for any given integer β, the highest β dimensions $E_{\nu-1},\ldots,E_{\nu-\beta}$ are used only during the last β merging phases. We could then depoly 2^{β} "small" CCC's, each with $n/2^{\beta}$ processors, to execute the first $\nu-\beta$ phases of merging in parallel, and subsequently supply the intermediate results to a "large" CCC, with n processors, to complete the execution of the sorting algorithm. The advantage of this strategy is that the smaller machines have short cycles, and work faster, while a large CCC would have to use its full cycle length in *all* stages of the algorithm. The transfer of results from the small CCC to the large CCC can actually be accomplished with *no* data movement by simply reconfiguring the network. Indeed, the reconfiguration of the 2^{β} small CCCs to the large one can be realized by suitably embedding the former into the latter. We first lay out the 2^{β} $(\frac{s}{2}\times\frac{r}{2^{\beta-1}})$-CCCs in a $2\times 2^{\beta-1}$ array (see Figure 6.5).

Note (again refer to Figure 6.5) that each cycle of a CCC in the lower tier faces at its upper end the lower end of a cycle in an upper tier CCC. Next we modify the layout by merging each pair of facing cycles into a single cycle with s processors and by using the last available $\beta-1$ rows of the top tier CCCs to realize the lateral connections for $E_{\nu-\beta-1},\ldots,E_{\nu-1}$ (see Figure 6.4). We obtain an $s \times t$ array, where in each column we have to provide a suitable switch to reconfigure the two original length-$s/2$ cycles into a single length-s cycle.

In the machine we have just described, the highest β -1 dimensions are lateral, but the β -th one is a cycle dimension. The next $\tau - \beta + 1$ dimensions are again lateral and the remaining $(\sigma - 1)$ ones are cycle dimension. For the first $\nu - \beta$ merging phases of the sorting paradigm, the 2^β CCCs are decoupled and work in parallel. Consider now phase $M_{\nu-\alpha}$ $(1 \leqslant \alpha \leqslant \beta)$, which corresponds to the execution of the sequence $E_{\nu-\alpha},\ldots,E_0$. The (lateral) dimensions $E_{\nu-\alpha},\ldots,E_{\nu-\beta+1}$ and the cycle dimension $E_{\nu-\beta}$ are executed using the full cycle length; next, the cycles are reconfigured to half length, and $E_{\nu-\beta-1},\ldots,E_0$ are executed in the "small" CCC's.

Before proceeding further, we consider the permissible values of the parameters β, s, and t. Since the top $s/2$ rows of the full network must support τ lateral dimensions, (i.e., in each cycle there must be at least one processor per dimension), we have:

$$\tau \leqslant \frac{s}{2}. \qquad 6.7$$

Since $(\beta - 1)$, the number of lateral dimensions connecting the "small" CCCs in Figure 6.5, must satisfy $1 \leqslant (\beta - 1) \leqslant \tau$ we trivially have:

$$2 \leqslant \beta \leqslant \tau + 1. \qquad 6.8$$

We shall now complete the modification of the CCC by fully exploiting the key idea which led to the network of Figure 6.5. This is achieved by defining as an $(s \times t)$-*pleated CCC* (PCCC) the network of Figure 6.5 where each of the 2^β component networks is itself a recursively defined $(\frac{s}{2} \times \frac{t}{2^\beta - 1})$-PCCC (rather than a conventional CCC). Note that β is a design parameter.

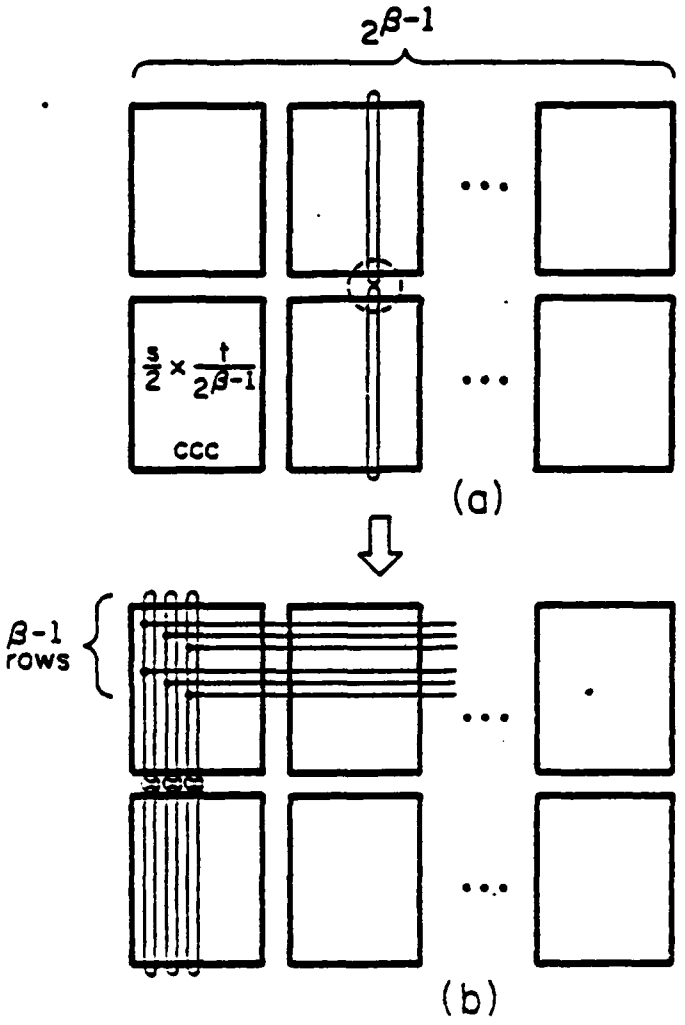


Figure 6.5. Construction of the PCCC. (a) Arrangement of 2^{β} independent small CCCs; (b) their interconnection to form the CCC.

An $(s \times t)$-PCCC with given β can be viewed as an $s \times t$ array of processors P_{ij} (as before, $s=2^{\sigma}$, $t=2^{\tau}$, $\sigma \geq \tau$, $0 \leq i < s$, $0 \leq j < t$) whose columns are organized as reconfigurable cycles, and

whose rows support lateral connections. Starting from the highest dimension $E_{\nu-1}$, lateral and cycle dimensions are interleaved, $(\beta-1)$ to one. This interleaving continues until we reach cycle dimension $E_{\nu-\lambda\beta}$, where the cycle length, $2^{\sigma-\lambda}$, is still adequate to accommodate the τ lateral dimensions. Thus, from the condition $2^{\sigma-\lambda} \geqslant \tau$ we obtain

$$\lambda \leqslant \sigma - \left\lfloor \log_2 \tau \right\rfloor \qquad 6.9$$

and λ is called the *depth of interleaving*. At this point the remaining $\nu - \lambda\beta$ dimensions are assigned as follows: the higher $\tau - (\beta-1)\lambda \geqslant 0$ are lateral, and the lower $\sigma - \lambda$ are cycle dimensions. Obviously we have

$$\beta \leqslant 1 + \frac{\tau}{\lambda}, \qquad 6.10$$

and the cycles are reconfigurable to any length 2γ where $\sigma - \lambda \leqslant \gamma \leqslant \sigma$. Note that there are $2^{\lambda} - 1$ reconfiguring switches per column.

An (8×16)-PCCC with $\beta = 3$ is illustrated in Figure 6.6. Notice that conditions 6.7 and 6.8 are automatically satisfied. Moreover, from 6.9 the maximum permissible value of λ is 1, whence condition 6.7 is confortably satisfied. Incidentally, note that, for the same value $\beta = 3$, the smallest PCCC with $\lambda = 2$ has 256 processors ($t = 16, s = 16, \beta = 3$).

Due to the above interleaving of dimensions, processor $P_{i,j}$ in the PCCC-array corresponds to cube-processor $P_{h'}$, where $h = j2^{\sigma} + i$ and h' is the integer obtained by permuting the binary representation of h according to the above interleaving scheme. This is illustrated in Figure 6.7.

Performance Analysis. In this section we give an upper bound to the area of the PCCC and to the time used to execute bitonic sorting. The PCCC is to be laid out in the rectangular grid. We will assume that a PCCC processor is endowed with a serial comparator and $O(k)$ bits of storage so that it fits in an $O(1)\times O(k)$ area. We also assume that edges have unit width. Data transmission takes place in serial fashion. Then the width of the $(s \times t)$-PCCC is easily seen to be $O(t)$, if we lay out each cycle in a constant number of vertical tracks. It is easy to see that an array row associated with the α—*th* highest lateral dimension ($\alpha = 1,\ldots,\tau$) (no matter what is the index of the dimension in the cube) is

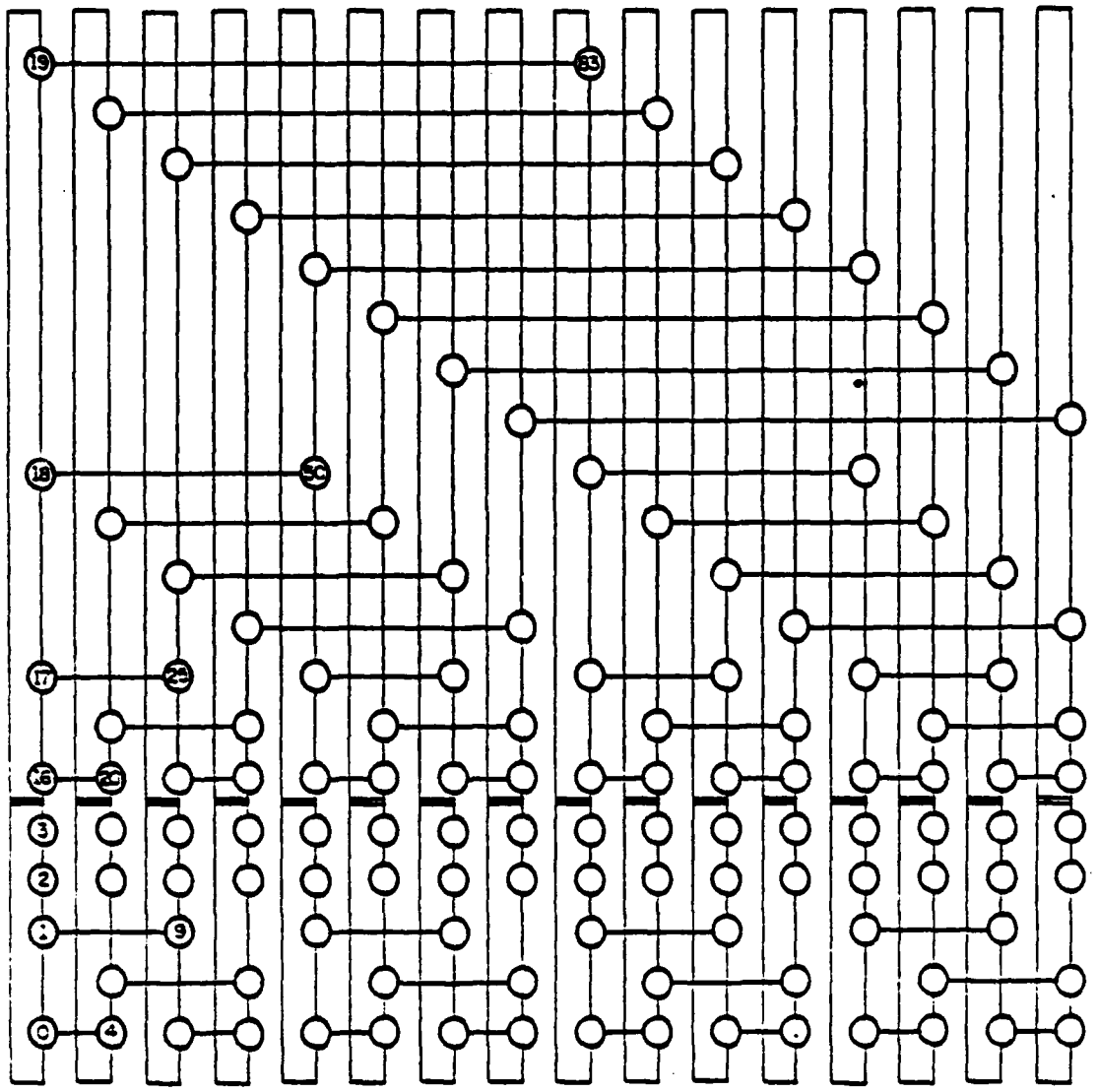


Figure 6.6. An (8 × 16)-PCCC with β=3. Since $\lambda = 1$ we have one reconfiguring switch per cycle.

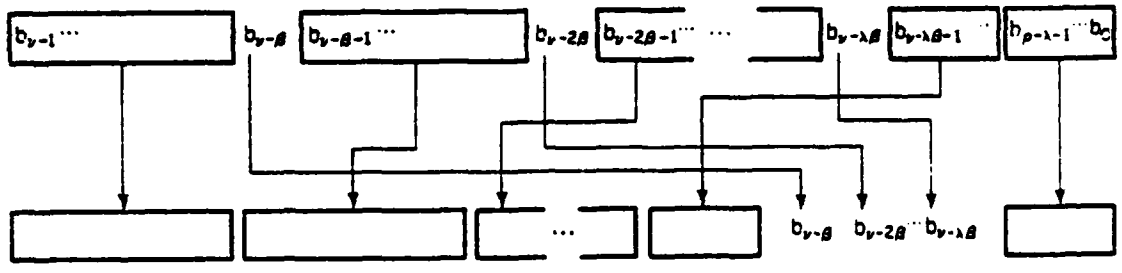


Figure 6.7. Bit position permutation induced by the pleating scheme (the arrangements of the h' and h are above and below, respectively.)

laid out with $t/2^{\alpha}$ tracks. When we consider the multiplicity of each dimension, i.e. the number of rows associated with it as a result of "pleating", we obtain the following formula for the height of the PCCC:

$$height = t\left(\left\lfloor\frac{1}{2}+\cdots+\frac{1}{2^{\beta-1}}\right\rfloor+2\left\lfloor\frac{1}{2^{\beta}}+\cdots+\frac{1}{2^{2(\beta-1)}}\right\rfloor+\cdots+2^{\lambda-1}\left\lfloor\frac{1}{2^{(\lambda-1)(\beta-1)+1}}+\cdots+\frac{1}{2^{\lambda(\beta-1)}}\right\rfloor\right.$$
$$\left.+\,2^{\lambda}\left\lfloor\frac{1}{2^{\lambda(\beta-1)+1}}+\cdots+\frac{1}{2^{\tau}}\right\rfloor\right)+O(ks).$$

By evaluating this sum, for $\beta = 2$ we have $height = O(\lambda t + ks)$, while for $\beta > 2$ we have $height = O(t + ks)$. Since in the case $\beta > 2$ the height does not depend upon λ, hereafter we further restrict β to be ≥ 3. Moreover, we add the condition $s \leq \sqrt{n/k}$, so that height $= O(t)$. We then conclude that $A = O(width \times height) = O(t^2)$.

The analysis of the computation time requires some additional discussion. A cycle dimension associated with arrays of length l uses $(l\text{-}1)$ steps, by the technique explained in Section 5.3.1. The execution of a set of consecutive lateral dimensions on a cycle of length l requires no more than $4l$ steps ($2l$ comparison exchanges and $2l$ shifts of the cycles). For our convenience we consider first the highest $\lambda\beta$ dimensions, and we group them in λ sets of size β, which are executed $\beta, 2\beta, \ldots, \lambda\beta$ times respectively. Since the execution of each set requires $\leq 4l$ steps, with $l = r/2^i$, for the i-th group ($i = 0, \ldots, \lambda-1$) we can upper bound the total number of steps τ_1 for these dimensions as follows:

$$\tau_1 \leq 4s\beta \left| 1 + \frac{2}{2} + \frac{3}{4} + \cdots + \frac{\lambda}{2^{\lambda-1}} \right| < 16s\beta.$$

The remaining dimensions are handled by a set of $(\frac{s}{2^\lambda} \times \frac{t}{2^{\beta\lambda}})$-CCCs, which have a cycle length $\theta(\log n)$ (since $\frac{s}{2^\lambda} = 2^{|\log_2 \tau|} = \theta(\tau) = \theta(\log n)$). Thus, referring to the well-known behavior of the conventional CCC, we know that the entire set of the last $\nu - \lambda\beta$ dimensions can be executed in $O(\log n)$ steps. On the other hand, the entire set is executed $O(\log n)$ times, thus they globally require $\tau_2 = O(\log^2 n)$ steps. Finally, recalling that k is the operand length, the total computation time T is given by $k(\tau_1 + \tau_2) = O(k(s + \log^2 n))$, and, for $s = \Omega(\log^2 n)$, we have $T = O(ks)$.

We can summarize the preceding discussion as follows.

Theorem 6.2. The pleated CCC can sort keys of length k in time $T = O(ks)$ and area $A = O(n^2/s^2)$ for any s in the range $[\Omega(\log^2 n), O(\sqrt{n/k})]$. For $k = \log n + \theta(\log n)$, the performance is $AT^2 = \theta(n^2 \log^2 n)$ for $T \in [\Omega(\log^3 n), O(\sqrt{n \log n})]$.

The PCCC has been first proposed in [BP84a], where a detailed description of the control structure is also given.

6.2.7 The Mesh-of-CCC

Another network that can execute the bitonic sorting paradigm with the same area-time performance of the PCCC is the mesh-of-CCC (MCCC), a suitable "hybridization" of mesh and CCC.

An (N,m)-MCCC, with $N = 2^{\nu}$, $m = 2^{\mu}$, and $t \triangleq N/m^2 = 2^{\tau}$ $(\tau = \nu - 2\mu)$ consists of m^2 CCC modules, each with t cycles of length τ. The Nm processors of the MCCC are conveniently indexed as

$$P_{i,j}^{p,q} : 0 \leq i,j < m, \quad 0 \leq p < \tau, \quad 0 \leq q < t. \tag{6.11}$$

For a fixed pair (i,j) the set $\{P_{i,j}^{p,q} : 0 \leq p < \tau, 0 \leq q < t\}$ is connected as an $(\tau \times t)$-CCC, and, for a fixed q, the set of processors $\{P_{i,j}^{p,q} : 0 \leq i,j < m\}$ is mesh connected (with i and j as row and column indices, respectively).

The MCCC graph can be laid out in a square of area $A = O(N^2/m^2)$, since each CCC requires $O(N^2/m^4)$ area, and channels of width $O(N^2/m^2)$ allow a straightforward implementation of mesh connections.

We discuss now the properties of the MCCC as an emulator of the binary cube. To avoid possible confusion, let us immediately say that the CCC modules will not be deployed in the standard mode described in Section 5.3.1. In fact a $(\tau \times t)$-CCC will be used to process only t (rather than $\tau \times t$) data items. This is accomplished by storing the data in row(0) (i.e., processors $P_{i,j}^{p,q} : 0 \leq q < t$, for fixed i and j) and by sending the data through the cycle for execution of the dimensions. Only the τ lateral dimensions of the CCC are then used, and therefore the length of the cycle τ does not need to be a power of two. An (N,m)-CCC will emulate a ν-dimensional binary-cube whose processors are $P(0), P(1), \ldots, P(N-1)$. We establish the following correspondence between MCCC processors and cube processors:

$$P_{i,j}^{p,q} \longleftrightarrow P_h, \quad h = j\,N/m + i\,N/m^2 + q. \tag{6.12}$$

It is easy to see that dimensions $E_0, \ldots, E_{\tau-1}$ are assigned to the CCC modules, dimensions $E_{\tau}, \ldots, E_{\tau+\mu-1}$ are assigned to the mesh columns, and finally dimensions $E_{\tau+\mu}, \ldots, E_{\tau+2\mu-1}$ are assigned to the mesh rows. Applying the by now familiar techniques for emulating the cube with a CCC or a

linear array, an Ascend (or Descend) algorithm can be executed in $O(\tau+m)$ word steps. On operands of length k, with bit-serial transmissions and operations, the computation time is $T = O((\tau+m)k)$. For m in the range $[\Omega(logN), O(\sqrt{N/logN})]$, considering that $\tau = O(logN)$, we obtain $T = O(mk)$.

In conclusion, for Ascend and Descend, the MCCC achieves $AT^2 = O(N^2k^2)$, for $T \in [\Omega(k\ logN), O(k\sqrt{N/logN})]$, which is optimal. (A variant of this result has been proved by [A83] for a network similar to the MCCC.)

It would not be difficult to see that the MCCC, in the form just described, does not achieve optimal performance when executing the Bitonic Sorting paradigm. However, it is also easy to realize that the problem lies in the assignment of the topmost 2μ dimensions. As we have already seen for the ordinary mesh, the best strategy consists in an alternate assignment of these dimensions to columns and rows. Formally, if

$$i = \sum_{h=0}^{\mu-1} i_h 2^h,\ j = \sum_{h=0}^{\mu-1} j_h 2^h,\ h' = \sum_{h=1}^{\mu-1} (2i_h + j_h)2^{2h}$$

we then establish between the processors of the MCCC and those of the cube the correspondence:

$$P_{i,j}^q \longleftrightarrow P_h,\ h = h'N/m^2 + q. \qquad 6.13$$

With this correspondence, dimensions $E_0, \ldots E_{\tau-1}$ of the binary cube are assigned to the CCC modules, dimensions $E_\tau, E_{\tau+2}, \ldots$ are assigned to the mesh rows, and dimensions $E_{\tau+1}, E_{\tau+3}, \ldots$ are assigned to the mesh columns. When executing the bitonic sorting paradigm $O(\tau logn)$ word steps are used by the CCCs. In fact there are $\nu = logn$ merging phases, and each of them involves no more than τ CCC dimensions. As for the mesh dimensions, they are used exactly in the same way as in a bitonic sorting algorithm on the mesh, and therefore their execution takes $O(m)$ word steps. Globally, $O(m + \tau\ logN)$ word-steps are needed. Since $\tau = O(logN)$, for operands of length k and for $m \in [\Omega(log^2N), O(\sqrt{N/k})]$ we obtain $T = O(mk)$. Recalling that $A = O(N^2/m^2)$ we have proved the following theorem. (The number of keys n equals the parameter N of the MCCC.)

Theorem 6.3. The mesh-of-CCC can sort n keys of length k in time $T = O(km)$ and area $A = O(n^2/m^2)$, for any m in the range $[\Omega(log^2n), O(\sqrt{nk})]$. For $k = logn + \theta(logn)$, the

performance is $AT^2 = O(n^2 \log^2 n)$, for $T \in [\Omega(\log^3 n), O(\sqrt{n \log n})]$.

We have already obtained three optimal sorters implementing the bitonic algorithm and respectively based on the mesh, the pleated CCC, and the mesh-of-CCC. It is indeed possible to construct several other optimal emulators of the binary cube, by suitable combinations of known emulators (for example the shuffle-exchange can be used to define the mesh-of-shuffles, or the shuffle-of-meshes, or the shuffle-connected-cycles). Although a systematic classification of the emulators of the cube is a problem interesting in its own right, it would not shed further light on bitonic sorting, and therefore we do not pursue it here.

However, there is one aspect of the PCCC and of the MCCC which is not satisfactory, namely that they do not achieve computation times smaller that $\Omega(\log^3 n)$, while the only obvious lower bound is $\Omega(\log^2 n)$, since there are $(\log n + 1)\log n/2$ consecutive steps in the bitonic sorting paradigm.

The discrepancy is obviously due to the fact that a bit-serial mode is adopted both for transmission and comparison-exchange operations. In fact, we can speed up the execution of bitonic sorting by resorting to parallel comparison-exchange, but - for $k = \theta(\log n)$ - each comparison step requires $\Omega(\log k) = \Omega(\log \log n)$ time, so that the global sorting time is still $\Omega(\log^2 n \log \log n)$.

To circumvent this difficulty we need to apply the pipeline principle not only to the words of a given sequence, but also to the bits of a given word. Prior to modifying the MCCC according to this idea, we discuss a mode of operation of the CCC network, which we call the *bit-pipeline* mode is contrast with the standard mode, which we call the *word-pipeline* mode.

For concreteness, we shall illustrate the bit-pipeline mode in the case of the bitonic merge algorithm, which is indeed a Descend algorithm where the operation steps consist in comparison-exchanges.

To sort a bitonic sequence of size $n = 2^\nu$ we display a $(\nu \times 2^\nu)$-CCC with $\nu 2^\nu$ processors P_j^i, $(0 \leq i < \nu,\ 0 \leq j < n)$. All processors in row(0) (i.e., $P_0^0, P_0^1, \ldots, P_0^{n-1}$) are also equipped with a shift register capable to store k-bit operands. All the edges are realized with unit bandwidth, and data transmission is serial.

A bitonic vector ([Ba68]) $A[0], \ldots, A[n-1]$ is initially input with component $A[j]$ loaded in P_j. Then, at each dimension $E_{\nu-1}, \ldots, E_1, E_0$ pairs of elements are compared and, if necessary, exchanged to place the smaller of the two in the cycle with the smaller number. More specifically, each processor reads the inputs starting from the *most significant bit* and compares them. As long as the two inputs agree, they are transmitted to the next processor in the same cycle. As soon as a discrepancy is detected, a switch is set and, from then on, the remaining substrings of each operand follow a fixed path, independently of their value.

For operands of k bits, the algorithm takes $O(k+\nu)$ units of time, in contrast with the $O(k\nu)$ units of time used in word-pipeline mode. If $k = O(\log n)$, as for the keys considered in this chapter, then $T = O(\log n)$.

Let us now consider an (n,m)-MCCC for the execution of the bitonic sorting paradigm, where the CCC modules function in the bit-pipeline mode. The only difference in performance is that the first τ dimensions ($\tau = \log n - 2\log m$) use $O(\tau+k)$ steps, each time that a group of them is executed, namely for each merging phase. Thus, since $\tau = O(\log n)$, $k = O(\log n)$, and the merging phases are $\nu = \log n$, the CCC dimensions $E_0, \ldots, E_{\tau-1}$ globally take $O(\log^2 n)$ time. Nothing is changed for the mesh dimensions $E_\tau, \ldots, E_{\nu-1}$, which take $O(m \log n)$ time, so that, for the entire algorithm, $T = O(\log^2 n + m \log n)$.

By considering m in the range $[\Omega(\log m), O(\sqrt{m/\log n})]$, we have then proved the following theorem.

Theorem 6.4. The mesh-of-CCC can sort in keys of length $k = \log n + \theta(\log n)$ with $AT^2 = O(n^2 \log^2 n)$ for $T \in [\Omega(\log^2 n), O(\sqrt{n \log n})]$.

We have now exhausted the potential of bitonic sorting. To obtain faster sorters we have to consider other algorithms.

6.3 NETWORKS FOR MERGE-ENUMERATION SORTING

In this section we concentrate on "very fast" VLSI sorters. The main objective is to design sorters with minimum running time $T = \theta(\log n)$. To achieve area-time optimality, these sorters must have area $A = \theta(n^2)$.

The first $\theta(\log n)$ time VLSI sorter has proposed by [L81] and [NMB83], and it is based on the Muller-Preparata algorithm executed by the orthogonal-tree (OT) network. We briefly review it in the sequel.

6.3.1 The Orthogonal-Tree Sorter.

To sort n keys $X_0, \ldots, X_{n-1}$ of length k, let us consider an OT network as the one discribed in Section 5.3.2. The Muller-Preparata algorithm (see Section 5.2.3) is executed as follows.

1. Key X_i is input at the root of row tree RT_i, and broadcast to the leaf processors $P_i^0, P_i^1, \ldots, P_i^{n-1}$ $(i = 0,1, \ldots, n-1)$.

2. Processor P_j^j sends its context X_j to the root of column tree CT_j which in turn broadcasts it to the leaf processors $P_0^j, P_1^j, \ldots, P_{n-1}^j$ $(j = 0,1, \ldots, n-1)$.

3. Processor P_i^j - which we assume to be equipped with $O(k)$ bits of memory, and with a serial comparator - compares X_i and X_j, and produces a bit $C_{i,j}$. $C_{i,j}$ is one if $X_i > X_j$ or if $X_i = X_j$, and $i > j$, and $C_{i,j}$ is zero otherwise $(i,j = 0,1, \ldots, n-1)$.

4. The internal nodes of row tree RT_i - which we assume to be equipped with a serial adder with a one-bit delay feedback on the carry - compute the sum $C_i = \sum_{j=0}^{n-1} C_{i,j}$. The sum is indeed produced at the root and will then be broadcast to all the leaves. Obviously C_i is the rank of X_i in the sorted output $(i = 0,1, \ldots, n-1)$.

5. Processor P_i^j compares C_i with j. If $C_i \neq j$, P_i^j remains idle. If $C_i = j$, P_i^j sends its content X_i to the root of tree CT_j.

6. The root of CT_j can now output the number received from the leaf, which will be Y_j (as usual $Y_0, Y_1, \ldots, Y_{n-1}$ is the sorted sequence corresponding to multiset $X_0, X_1, \ldots, X_{n-1}$).

Both operations and data transmission are done bit-serially, so that all the edges of the OT can be realized with unit bandwidth. It is easy to see that each step of the algorithm takes at most $O(k + \log n)$ time. Thus, for $k = \log n + \theta(\log n)$, the global running time is $T = \theta(\log n)$.

The OT is area-time suboptimal because its area is $A = O(n^2 \log^2 n)$. However, it is an interesting network, and it is also useful as a building block of optimal networks, as we shall see in the following sections.

6.3.2 A Network for the Merge-Enumeration Combiner

We now turn our attention to the class of merge-enumeration combine-sort algorithms (Section 5.2.3). Since the original description of these algorithms [P78] is related to the shared-memory machine, we need to investigate possible implementations with finite degree networks.

We begin by proposing a parallel network for the fundamental block of the sorter, namely the (ml)-combiner, where we will assume that $m = 2^\mu$ and $l = 2^\lambda$ are powers of two. This network will accept as input m sorted sequences of l elements each,

$$S_i = (s_i(0), s_i(1), \ldots, s_i(l-1)), \quad i = 0,1,\ldots,m-1$$

and produce as output a single sorted sequence S, which is the combination of $S_0, \ldots, S_{m-1}$, and has $L = m\,l = 2^\lambda$ elements

$$S = (s(0), s(1), \ldots, s(L-1)).$$

The (ml)-combiner will execute the algorithm based on pairwise merging as outlined in the preceding section. Its organization is illustrarted in Figure 6.8. It consists of m^2 modules (each capable of merging two sequences of length l and of computing partial ranks), laid out as a square $m \times m$ mesh and indexed as $M_{i,j}$ $(i, j = 0,1,\ldots,m-1)$. The modules of each row are interconnected as the leaves of a binary tree of bandwidth l; so are the modules of each column. Thus, the combiner has the structure

of the orthogonal-trees machines, whose leaves are merging modules. The interconnecting trees have the following functions:

(1) to "broadcast" a sequence to all units in which it must be merged with some other sequence;

(2) to compute global ranks from partial ranks;

(3) to rearrange the elements according to their ranks into the sorted sequence S.

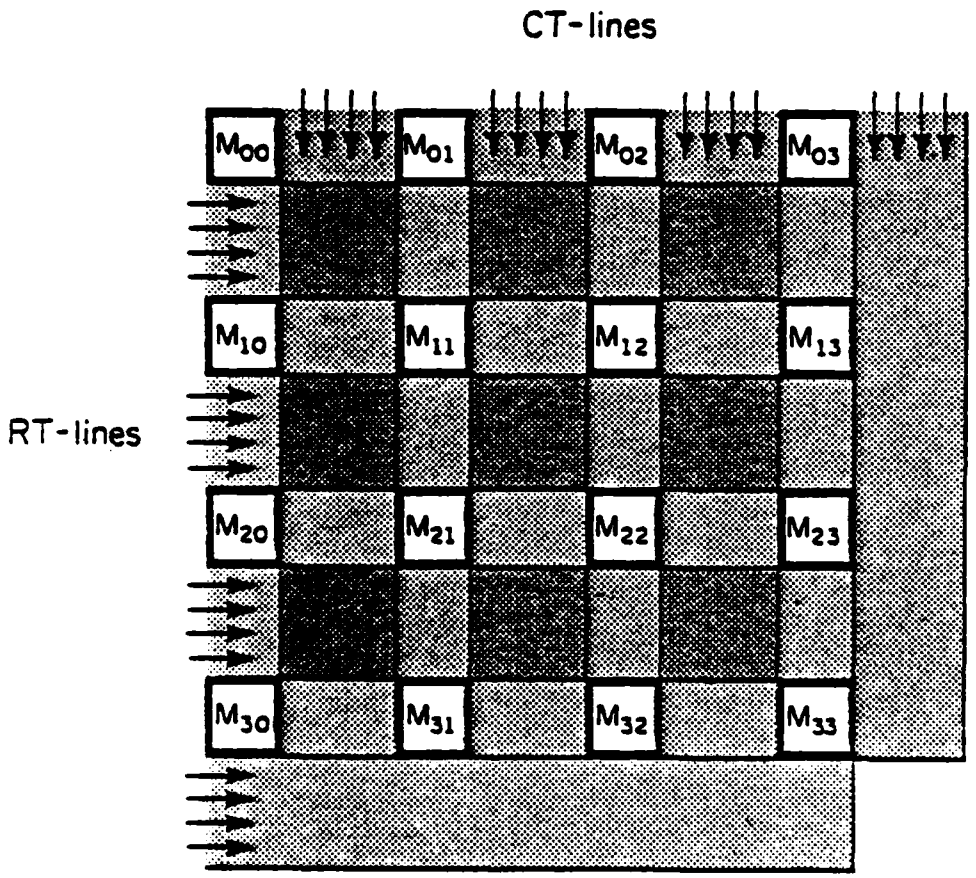


Figure 6.8. Overview of (mh)-COMBINER, for $m = 4$.

We will now describe in some detail the *merging modules* and the *interconnecting trees.*

Merging modules. Merging module $M_{i,j}$ will merge sequences S_i and S_j and compute $C_{i,j}(h)$, for $h = 0, \ldots, l\text{-}1$. We recall that $C_{i,j}(h)$ is the number of elements of S_j that are less than (respectively less than or equal) $s_i(h)$ when $i \leqslant j$, (when $i > j$). Each module is realized as a $(\lambda + 1 \times 2^{\lambda+1})$-CCC (See Figure 6.9.) We shall refer to the processors of module $M_{i,j}$ as micromodules and we shall index them as $P_{i,j}^{p,q}$, with $0 \leqslant p < t + 1$, and $0 \leqslant q < 2^{\lambda+1}$.

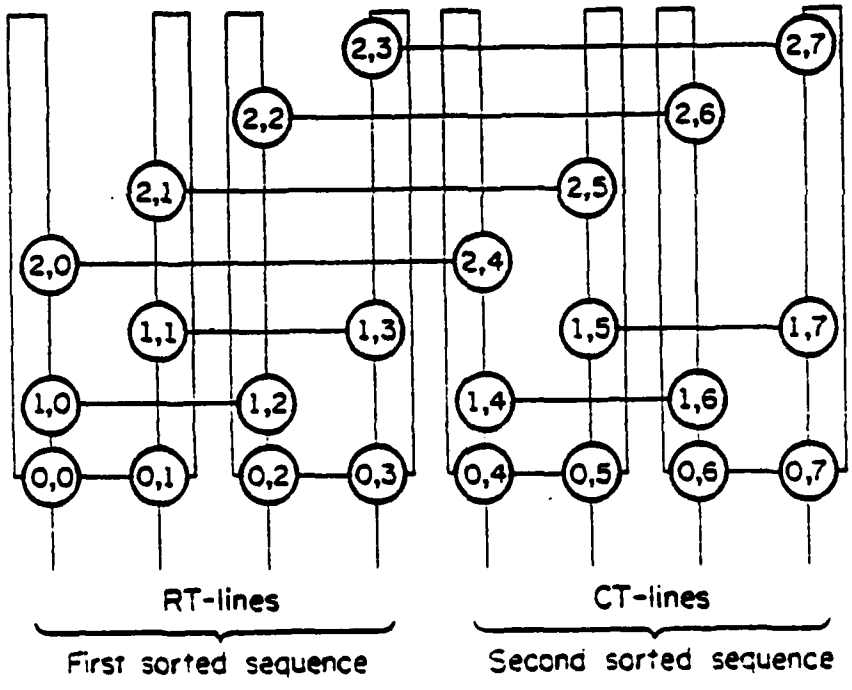


Figure 6.9. Merging unit $M_{i,j}$ realized by a $(3,2^3)$-CCC, used to merge two sequences with four elements each.

The layout area of a merging module is of order $O(h^2)$ (Section 5.3.1).

Interconnecting trees. As indicated earlier, the merging modules are interconnected by two families of $L = ml$ complete binary trees with $m = 2^\mu$ leaves and bandwidth 1. We will refer to these families as the row trees and column trees.

The lines of the row trees and the column trees are respectively labelled $RT_i(h)$ and $CT_i(h)$, $i = 0, \ldots, m-1$; $h = 0, \ldots, l-1$. The trees and the merging modules are connected through a small interface, whose structure will be fully specified in connection with the description of the combination algorithm in the next section. At this point we just say that the leaves of $RT_i(h)$ are, from left to right, connected to the CCC micromodules $P_{i,0}^{0,h}, P_{i,0}^{1,h}, \ldots, P_{i,0}^{m-1,h}$; the leaves of $CT(h)$ are connected to the CCC micromodules $P_{0,0}^{jl-1+h}, P_{1,0}^{jl-1+h}, \ldots, P_{m-1,0}^{jl-1+h}$; in other words, the row trees and the column trees are respectively connected to the RT and CT lines of the merging modules. The connection between each leaf of a tree and the coresponding CCC micromodule is realized through a buffer register of the appropriate size (adequate to store one element to be sorted). The situation is illustrated in Figure 6.10.

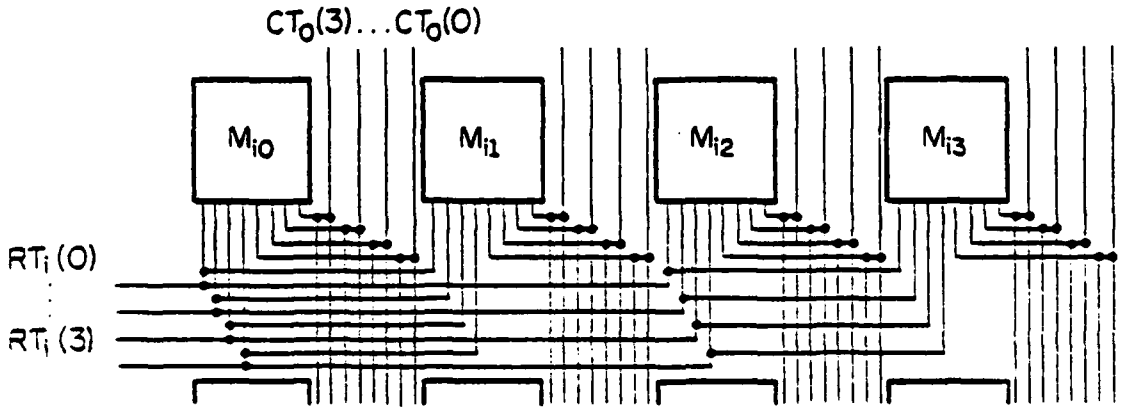


Figure 6.10. Interconnection of modules and trees.

6.3.3 The Combination Algorithm

We describe now how the merge-enumeration algorithm can be executed by the network introduced in the preceding section. For convenience we split the algorithm into several phases.

(A) *Input of Data and Broadcasting to Merging Modules*

Element $s_i(h)$ is input at the root of tree $RT_i(h)$, and is then broadcast to all leaves of the tree. At this point, the left half of row(0) in module $M_{i,j}$ contains the sequence S_i. To fill the right halves of row(0) of all modules, we proceed as follows. First, in each "diagonal" module M_{ii} the sequence S_i is copied in the second half of row(0). (This can be done by using the connection of row(λ) between the left and the right half of the machine.) Next, from micromodule $P_{j,0}^{l-1+h}$, which is a leaf of $CT_j(h)$, element $s_j(h)$ is broadcast (through the root) to all other leaves of the same tree. At this point, the merging module $M_{i,j}$ contains S_i and S_j in row(0) and merging can begin.

(B) *Merging and Partial Rank Computation*

Merging can be executed by resorting to the bitonic algorithm, and using the CCC modules in a bit-pipeline node, as explained in Section 6.2.7. However, in order to execute bitonic merging, we first need to reverse the order of S_j. This is accomplished by an Ascend algorithm in which columns l to $2l$-1 of each $M_{i,j}$ exchange their data at dimensions $E_0, \ldots, E_{\lambda-1}$ while columns 0 to l-1, remain idle. All the columns are idle at dimension E_λ.

Now the data are ready and bitonic merging can be executed. At the end of merging, the result resides in row(0) of the CCC, and the element in $P_{i,0}^{l-1}$ $0 \leq h \leq 2l-1$, has rank h in merge (S_i, S_j). Now we want to transmit the ranks of $s_i(0), \ldots s_i(l-1)$ to processors $P_{i,0}^0, \ldots, P_{i,0}^{l-1}$, respectively. This is accomplished by retracing backwards the path traversed by each element $s_i(j)$, and is easily done if each $P_{i,0}^h$ keeps track of whether it exchanged or not the operands during the merging sprocess. So, all we have to do is to run the machine backwards, with an Ascend algorithm, which applies to the ranks the inverse of the permutation that merged the elements. At the end of this phase, processor $P_{i,0}^h$, $0 \leq h \leq l-1$, stores the number of elements in merge (S_i, S_j) that are less than $s_i(h)$.

If from this number we subtract h we obtain $C_{i,j}(h)$, the number of elements of S_j which are less than $s_i(h)$. We call the $C_{i,j}$'s *partial ranks* because from them we can compute the rank of each $s_i(h)$ in the sorted sequence S as $C_i(h) = \sum_{j=0}^{m-1} C_{i,j}(h)$.

(C) *Total Rank Computation*

It is immediate to see that at the end of phase B the partial ranks $C_{i,0}(h), C_{i,1}(h), \ldots, C_{i,m-1}(h)$ of $s_i(h)$ are available exactly at the leaves of row tree $RT_i(h)$. By having in each internal node of the tree a full adder with 1-bit delay feedback on the carry, we can then obtain at the root of RT_i the sum $C_i(h)$ of the values stored at the leaves. The nodes work as serial adders and the tree is used in a pipelined fashion, so that the time required is $O(\mu+\lambda)$, where $\mu = \log m$ is the depth of the tree, and $\lambda+1$ is the wordlength of the operands (note that $C_{i,j}(h) \leq 2^{\lambda}$). Within the same order of time, we can subsequently broadcast $C_i(h)$ from the root to the leaves. (Indeed $C_i(h) < 2^{\lambda+\mu}$, so it can be expressed by $\lambda + \mu$ bits.)

(D) *Sorting Permutation and Output of Data*

We want to output the elements $s(0), \ldots, s(L-1)$ of the sorted sequence from the roots of the column trees, and, specifically, we want the root of $CT_j(h)$ to output element $s(j2^{\lambda}+h)$. This corresponds to a natural right-to-left order of the column trees as they appear in the layout of Figure 6.10.

Considering a generic element $s_i(p)$ with rank $C_i(p)$, the binary spellings of the integers j and h so that $s_i(p)$ will emerge from the root of column tree $CT_j(h)$ are readily obtained by taking the μ most significant bits and the λ less significant bits of the rank $C_i(p)$ to represent h and j, respectively. Thus, as a first step, we "activate" in $M_{i,j}$ the elements of sequence S_i that have to emerge from trees CT_j's, and "inhibit" all other elements. The *active elements* are those whose rank $C_i(p)$ has the μ most significant bits agreeing with the column number j of the merging module. Next, we rearrange the active elements in $M_{i,j}$ so that $s_i(p)$ is sent to $P_{i,h}$, with $h = C_i(p) \bmod l$.

This operation is essentially a permutation of the active (and non-active) elements, and can be done by using the CCC as an emulator of the Benes network [Be64]. The setting of the switches, although nontrivial, is greatly simplified with respect to the general case by the fact that the active elements do not change their relative order. The desired rearrangement can be done by using the idea of *concentration* introduced in [NS82], and *expansion*, which could be viewed as the inverse of concentration. If t elements are active in the given module, they are first sent to the t leftmost colums of the CCC (concentration), and then routed to the destination columns (expansion). A straightforward adaptation of the algorithm that is proposed in [NS82] for concentration in the cube-machine shows that an Ascend and a Descend phase is all that is required to rearrange data on our CCC. Some bits required to set the switches must be precomputed. This task could be performed by the CCC, or (to keep the micromodule structure as simple as possible), the task can be assigned to a binary tree of full adders whose leaves would be contained in the interface between the CCC and the row-trees.

During the entire rearrangement task, computation takes place only in the left-half of the CCC without using dimension E_σ. We then transfer each active element from $P_{i,j}$ to $P_{i,j'}$ with a straightforward use of dimension E_λ.

At this point element $s(j2^\lambda + h)$ is in $P_{i,j'}$, (where the value of i is determined by the input sequence to which $s(j2^\lambda + h)$ originally belongs), and is ready to be transmitted to the root of $CT_j(h)$, where it is output.

Performance Analysis and Modification of the Network. Since both the CCCs and the interconnecting trees work in pipeline in bit-serial mode, any operation takes time proportional to the sum of the operand length and the pipe depth. For the CCC, the depth is $\lambda + 1$ and the operand length is either k (input words) or $\lambda + 1$ (partial ranks). Since a constant number of Ascend and Descend algorithms are executed, we conclude that $O(\lambda + k)$ total time is spent in the CCCs. For the trees the depth is $\mu + 1$, and the operand length is either k (input words) or $\lambda + \mu$ (total ranks). Since a constant number of fan-in and fan-out algorithms are executed, we conclude that $O(\lambda + \mu + k)$ total time is spent in the trees. Thus, the time spent in the interconnecting trees dominates that spent in the CCCs.

Recalling that a full binary tree on m aligned leaves is laid out in height $\theta(logm)$ and that there are l row and column trees, we conclude

Lemma 6.1. A full-tree $(2^{\mu},2^{\lambda})$-combiner of keys of length k can be laid out in a square of width $O(\mu 2^{(\lambda+\mu)})$ and operates in time $T = O(\lambda + \mu + k)$.

We now observe that when $k = \Omega(2^{\mu})$, then $T = O(\lambda + k)$. In this case the time performance of the trees is insignificantly degraded if we realize them as comb-trees, rather than as full binary trees. The depth increases from μ to 2^{μ} (which is tolerable in time since $2^{\mu} = O(k)$), but the layout area decreases by a factor of $O(\mu^2)$. We conclude:

Lemma 6.2. A comb-tree $(2^{\mu},2^{\lambda})$-combiner of keys of length $k = \Omega(2^{\mu})$ can be laid out in a square of width $O(2^{(\lambda+\mu)})$ and operates in time $T = O(\lambda + k)$.

Summary of Symbols for an (m,l)-Combiner

Sizes: $m = 2^{\mu}$, $l = 2^{\lambda}$, $L = ml$, $k = keylength$.

Input sequences:

$S_i = (s_i(0),s_i(1),\ldots,s_i(l-1))$ $i = 0,1,\ldots,m-1$.

Output sequence:

$S = (s(0),s(1),\ldots,s(L-1))$.

Merging modules: $(\lambda + 1,2^{\lambda-1})$-CCC's

$M_{i,j}$: $i,j = 0,1,\ldots,m$-1

$P_{i,j}^{p,q}$: $0 \leq p < \lambda + 1$, $0 \leq q < 2^{\lambda}$, micromodules of $M_{i,j}$.

Row-trees and column-trees:

$RT_i(h)$, $CT_j(h)$: $0 \leq i,j < m-1$, $0 \leq h < l-1$.

6.3.4 The Sorter

The combiner can be used to construct a general network for combination-sort. As an intermediate step in the construction, we introduce a new operation called *coalescence*. Given a collection of n elements, partitioned into n/l_{i-1} sorted subsequences each containing l_{i-1} elements, and given a multiple l_i of l_{i-1}, which is also a divisor of n, we call $(n; l_{i-1}: l_i)$-coalescence the operation of combining (in the sense defined earlier) consecutive blocks of $m_i = l_i/l_{i-1}$ sequences.

If we refer to the tree of Figure 5.1, we can easily see that each level of the tree corresponds to a coalescence of the input sequence. If we call coalescer a network that performs a coalescence, we can build a combination- sorter by cascading a suitable set of coalescers, as shown in Figure 6.11.

The coalescer. An $(n; l_{i-1}: l_i)$-coalescer can be easily constructed by using $n_i \triangleq n/l_i$ (m_i, l_{i-1})-combiners. Let us assume for simplicty, that n_i is a perfect square. We can then lay out the combiners in an $\sqrt{n_i} \times \sqrt{n_i}$ array with input and output lines running in a chosen direction, say, parallel to the rows.

To estimate the area of the coalescer, we first assume to use full-tree combiners, so that the side of the combiner has a length of $O(l_i \log m_i)$ (parallel to the rows). Using Lemma 6.1 we have

$$Height = O(\sqrt{n_i}\, l_i \log m_i + n_i l_i) = O\left(+\frac{\log m_i}{\sqrt{n_i}}\right)$$

$$Width = O(\sqrt{n_i}\, l_i \log m_i) = O\left(n\frac{\log m_i}{\sqrt{n_i}}\right).$$

An example with $n = 4$ is shown in Figure 6.12. The computation time is readily found as $T_F = O(\lambda + k + \log m_i)$. We conclude:

Lemma 6.3. An $(n; l_{i-1}: l_i)$ full-tree coalescer can be laid out in a rectangle $O((n/l + \log m_i/\sqrt{n_i})) \times O(n \log m_i/\sqrt{n_i})$ and operates in time $T_F = O(\lambda + k + \log m_i)$ (k is the input keylength, $n_i = n/l_i$, $m_i = n$). When $k \log m + \theta(\log n)$, then $T_F = O(\log n)$.

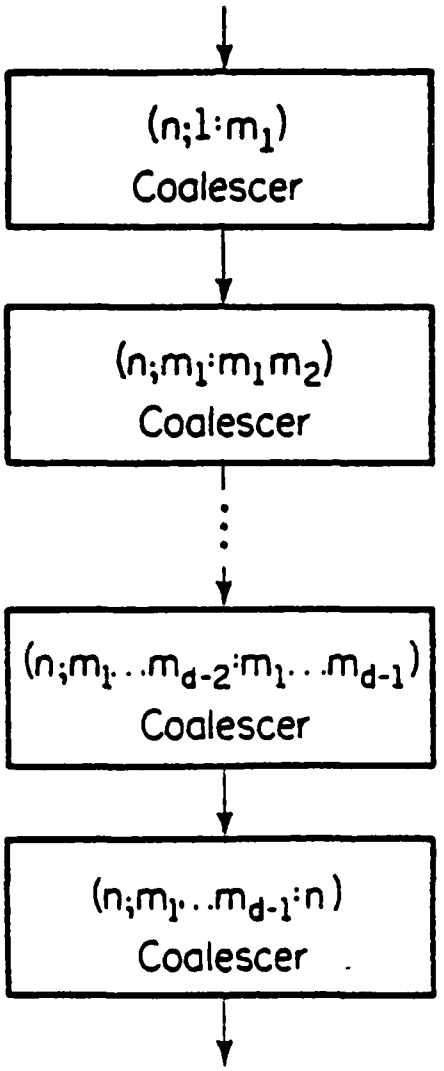


Figure 6.11. Combination-sorter as a cascade of coalescers.

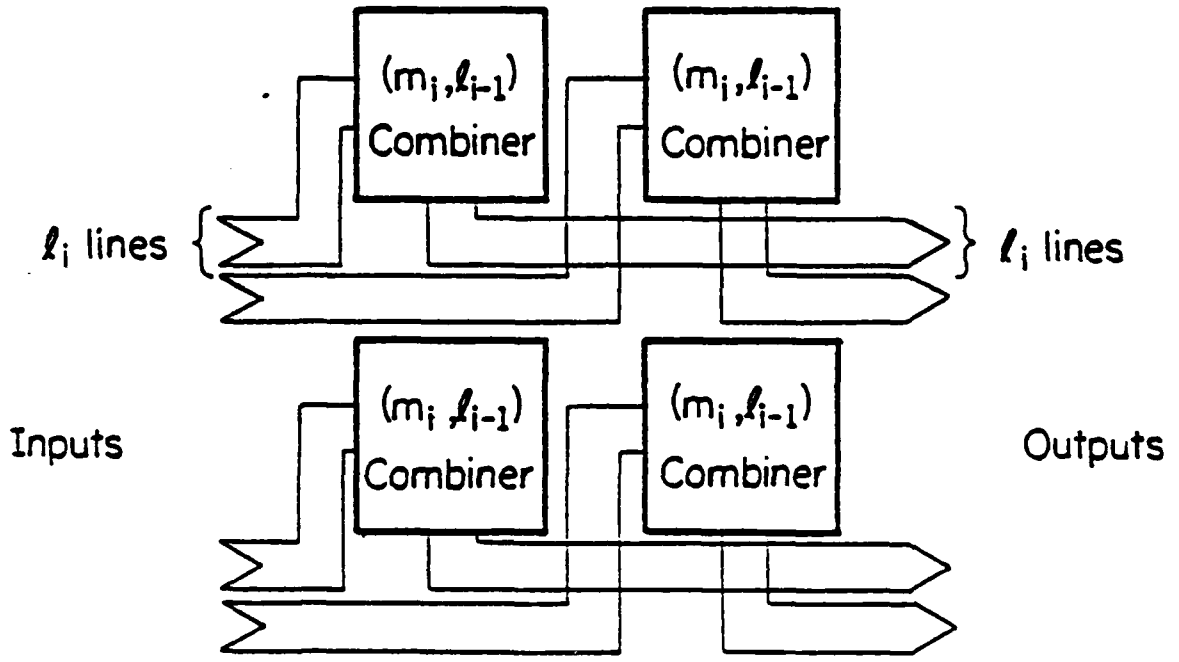


Figure 6.12. Layout of an $(n; l_{i-1}: l_i)$-coalescer with $n_i = n/l_i$ $(m_i = l_i/l_{i-1}, l_{i-1})$-combiner. (In the figure, $n_i = 4$.)

Similarly, Lemma 6.2 yields the following result:

Lemma 6.4. An $(n; l_{i-1}: l_i)$ comb-tree coalescer can be laid out in a rectangle $O(n) \times O(n \log m_i / \sqrt{n_i})$ and operates in time $T_C = O(\tau + k + m_i)$. If both k and m_i are $O(\log n)$, then $T_C = O(\log n)$.

An Optimal VLSI Sorter. We now show that there is a combination-sorter for keys of length $k = \log n + \theta(\log n)$ that sorts n elements in time $T = O(\log n)$ and area $A = O(n^2)$, thus achieving the known lower bound for this problem. The sorter we propose is given by the block diagram in Figure 6.13. By the previous Lemmas 6.3 and 6.4 we see that the coalescers can be laid out in area

(*width* × *height*) $O(n) \times O(n)$, $O(n\log\log n/\log n) \times O(n)$, and $O(n) \times O(n)$, respectively. It is also clear that the total time is $O(\log n)$. So we have:

Theorem 6.5. There is a VLSI merge-enumeration sorter of n keys of length $k = \log n + \theta(\log n)$ with area $A = O(n^2)$, and computation time $T = O(\log n)$.

Remark. The first coalescer stage of the sorter we have just described consists of $\log^2 n$ sorters, each processing a sequence of $n/\log^2 n$ keys. These sorters are essentially orthogonal-tree sorters of the type described in Section 6.3.1. Strictly speaking, for $l = 1$, the (n,l)-combiner of Section 6.3.2 consists of two families of binary trees $(RT_0(0),\ldots,RT_{m-1}(0))$ and $(CT_0(0),\ldots,CT_{m-1}(0))$ such that the j-th leaf of $RT_i(0)$ and the i-th leaf of $CT_j(0)$ are constructed (they indeed form the merging module M_{ij}), whereas in the OT network they would be identified.

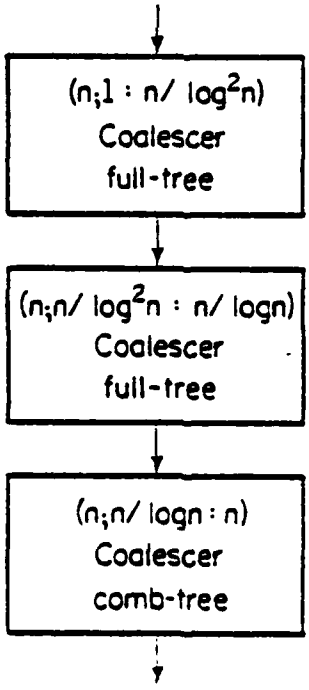


Figure 6.13. An optimal VLSI merge-enumeration sorter with three coalescers.

6.3.5 Sorting in Time $T \in [\Omega(\log n), O(\log^2 n)]$

We have seen that $AT^2 = \theta(n^2 \log^2 n)$ can be achieved for $T = \theta(\log n)$ (Theorem 6.5) and for $T \in [\Omega(\log^2 n), O(\sqrt{n \log n})]$ (Theorems 6.1, 6.2, 6.4). It is natural to try to extend the result to the interval $T \in [\Omega(\log n), O(\log n)]$. For this purpose we start from the following observation. A combine-sorter with n/s input can sort (in time $O(s \log n)$ the area $O(n^2/s^2)$) $s = 2^\sigma$ sequences of n/s elements each. These sequences can then be fed, say one per column, into an (m,s)-MCCC. At this point, the sequence in each CCC module is already sorted, and the MCCC is ready (after inverting the order of some sequenes to comply with bitonic sorting rules) to execute the last 2σ merging phases. (For the sake of simplicity we will ignore the fact that only σ phses would be really necessary after the work done by the combination-sorter.) A simple analysis allows us to conclude that, in the process, the MCCC executes $O(\log s + s)$ steps using $O(\log n)$ time for each, thus running for a total time $T = O(s \log n)$. We can then state:

Theorem 6.6. There is a VLSI sorter of n keys of length $k = \log n + \theta(\log n)$ with optimal $AT^2 = \theta(n^2 \log^2 n)$ for any computation time $T \in [\Omega(\log n), O(\sqrt{n \log n})]$.

With Theorem 6.6, the characterization of the area-time complexity of the $(n, \log n + \theta(\log n))$-sorting problem is complete (within multiplicative constant factors).

6.4 OTHER OPTIMAL NETWORKS

For completeness, we report here two other interesting results concerning optimal $(n, \log n + \theta(\log n))$-sorters.

The first result is due to Leighton [L84], and provides a design that achieves optimal $AT^2 = \theta(n^2 \log^2 n)$ for $T \in [\Omega(\log n), O(\sqrt{n \log n})]$. The network consists of a suitably interconnected family of OT-networks. The algorithm is a combine-sort of the hybrid type, with the first stage of combinations performed with the Muller-Preparata algorithm, and the remaining stages (one or two depending on T) performed with the multiway-shuffle algorithm.

We refer the reader to [L84] for more details. However, we shall return to Leighton's algorithm in Section 7.2, where we study circuits to sort keys of medium length.

The second result is due to Bilardi and Preparata [BP84c] who have shown that the AKS network [AKS83] can be optimally laid out in area $A = O(n^2)$, while maintaining a sorting time $T = O(\log n)$ on keys of $O(\log n)$ bits.

The details are rather intricate, and hence are not repeated here. An open question, as far as we know, is the existence of optimal networks that execute the AKS sorting algorithm in time greater than $\Theta(\log n)$. Obviously, the answer to this question would not improve the characterization of the area-time complexity of sorting, since we have already several optimal constructions, but could shed some light on the algorithm itself.

CHAPTER 7

SORTING KEYS OF ARBITRARY LENGTH

7.1 INTRODUCTION

In Chapter 6, we have studied in depth the (n,k)-sorting problem for the special, but important, case when $k = \log n + \theta(\log n)$. In this chapter we consider the general problem of sorting keys of arbitrary length.

The classification of keys into short ($k \leq \log n$), medium-length ($\log n < k < 2\log n$), and long ($2\log n \leq k$), introduced in Chapter 4 in the context of lower-bound arguments, maintains its validity when considering circuit constructions. Indeed, a different algorithm and a different network are appropriate to each of the above three intervals of key lengths.

The difference between the VLSI model and other models of parallel computation reveals its full extent in the present chapter, where an attempt to optimize the area-time performance of VLSI sorters leads to the formulation of novel sorting algorithms.

For short and medium-length keys the efficiency of the new algorithms is based on the use of the appropriate encoding schemes for the multisets being processed. For long keys the efficiency of the algorithms rests instead on the adoption of non word-local I/O protocols that induce a partition of the chip into regions within which primary flow is confined, and among which only secondary flow is exchanged.

All the algorithms we shall consider in this chapter make use at some stage, of an $(n,\log n + \theta(\log n))$-sorting procedure. Thus, the constructions confirm that the keylength $k = \log n + \theta(\log n)$ plays a special role, as a careful analysis of lower-bound arguments had already indicated in Section 6.1.

Sorting algorithms and networks for medium-length, short, and long keys are respectively discussed in the next three sections of the chapter.

7.2 SORTERS FOR KEYS OF MEDIUM LENGTH

In this section we derive upper bounds for the $(n, \log n + h)$-sorting problem for $0 < h < \log n$. We recall from Theorem 4.14 and 4.15 that

$$AT^2 = \Omega(n^2 h^2) \qquad 7.1$$

and, for boundary chips,

$$AT^2 = \Omega(n^2 h \log n) . \qquad 7.2$$

When $h = \theta(\log n)$, lower bounds 7.1 and 7.2 are of the same order, and they are both achieved by the constructions of Chapter 6. However, a careful analysis of the upper bounds reveals that they are of the form $AT^2 = \theta(n^2(\log n + h)^2)$, so that even if h is zero we have $AT^2 = \theta(n^2 \log^2 n)$. Thus, for $h = o(\log n)$, the sorters of Chapter 6 are slightly suboptimal.

In the following, we shall see that the performance of these sorters can indeed be improved by exploiting the fact that a multiset of n keys of length $k - \log n + h$ can be encoded with $2n(h+1)$ bits, as it has been shown in Section 2.2.4.

In the design of our sorter for keys of medium length, we shall use an approach very frequently adopted in the design of VLSI networks, which can be formulated as follows. Let Π be a problem amenable to a divide-and-conquer solution, and let us assume that we are trying to solve Π with target performance $AT^2 = O(n^2)$ on input instances of size n. Additionally, let us suppose that a design for Π is known with performance $A_0 T_0^2 = O(g(n) n^2)$, where $g(n)$ - a monotone increasing function of n - is the gap between the performance of the known design and the target. Then, if we decompose the problem into $g(n)$ subproblems of size $n/g(n)$, we can solve the subproblems with $g(n)$ networks of performance $(A_0(n/g(n)), T_0(n/g(n)))$, globally achieving

$$A_1 T_1^2 = g(n)\, O(g(n/g(n))\, n^2/g^2(n)) = O(n^2).$$

Thus, to obtain the desired result, we are left with the problem of combining the solutions to the $g(n)$ subproblems in area and time of the same order as A_1 and T_1 respectively.

This approach effectively transforms the task from the design of the entire system to the design of a subsystem for the combination step of the divide and conquer strategy. According to intuition, the better is the construction that we use for the subproblems, i.e. the smaller is the gap $g(n)$, the smaller is the number of subproblems that have to be combined, and therefore the easier is the combination step.

For the sorting problem we are presently considering, we already know several designs achieving $A_0 T_0^2 = O(n^2 \log^2 n)$, and we can try to follow the above approach. For concreteness, we refer to the boundary chip case, so that our target is a design with performance $AT^2 = O(n^2 h \log n)$, and $g(n) = \log n / h$. Thus, we can sort $\log n / h$ sequences of $nh / \log n$ elements each within an area-time performance allowed by our objective, and we are then left with the problem of combining these sequences.

For this combination we shall use Leighton's multiway-shuffle algorithm, for reasons that will be apparent as description of the sorter unfolds and that, at this point, we can informally explain as follows.

To attain the $AT^2 = \Omega(n^2 h \log n)$ lower bound we cannot afford to maintain the list representation of the input multiset throughout the entire algorithm. Indeed, this would imply an $\Omega(n \log n)$ information exchange across a suitable bisection of the network, whereas we can only afford an $O(nh)$ information exchange. Thus, it is essential to compactly encode the multiset, or some part thereof, in the stages of the algorithm that pose the heaviest demand in terms of global rearrangement of data.

We shall indeed use the insert-and-prune encoding scheme to solve this problem. On the other hand, when a multiset is compactly encoded, the individual elements are not easily accessible for operations, say, as comparison-exchanges, therefore it is very desirable to be able to use the compact form only for data transmission, and to recover the natural list representation wherever operations are to be executed.

The multiway-shuffle combination is ideally suited to our purposes, because the only global rearrangement of data occurs when shuffling and unshuffling the keys, while the other stages of the algorithm require data interactions only within the blocks of a suitable partition of the input multiset.

There is a difficulty, however. The insert-and-prune encoding is itself based on sorting a sequence of length twice as large as the one being encoded. Thus, using the sorters of Chapter 6, we cannot encode the entire input multiset at once without exceeding our target performance. Hence, encoding will be applied to suitable subsets of keys. However, this entails a loss in the efficiency of the encoding. (The reader will easily convince himself that the optimal encoding of $S_1 \cup S_2$ requires fewer bits than the sum of the number of bits required to encode S_1 and S_2 separately.) This difficulty will be at least partially overcome by resorting to a recursive technique.

We have presented the main ideas involved in the design of the sorter of medium-length keys, and we are ready to give a detailed description of the construction.

The ideas we have informally presented above will be combined according to the following scheme, illustrated in Figure 7.1, consisting of three basic steps:

1. Given a sorter design, we show how to constuct an encoder/decoder of multisets based on the insert-and-prune method. The area-time performance of the encoder/decoder will be a function of the performance of the sorter used in the construction.
2. Given designs of an encoder/decoder and of a multiway shuffler/unshuffler, we show how to construct another shuffler/unshuffler whose performance is better than the one of the original shuffler/unshuffler.
3. Given designs of a shuffler/unshuffler and of a sorter, we show how to construct a new sorter (with improved performance) by resorting to multiway-shuffle combination.

The scheme will be iteratively applied. The first stage of the iteration will use a sorter of performance $AT^2 = O(n^2 \log^2 n)$, and a straightforward implementation of shuffler and unshuffler with the same performance. Subsequent stages will use as a starting point the designs for the sorter and for the shuffler/unshuffler obtained in the previous stage.

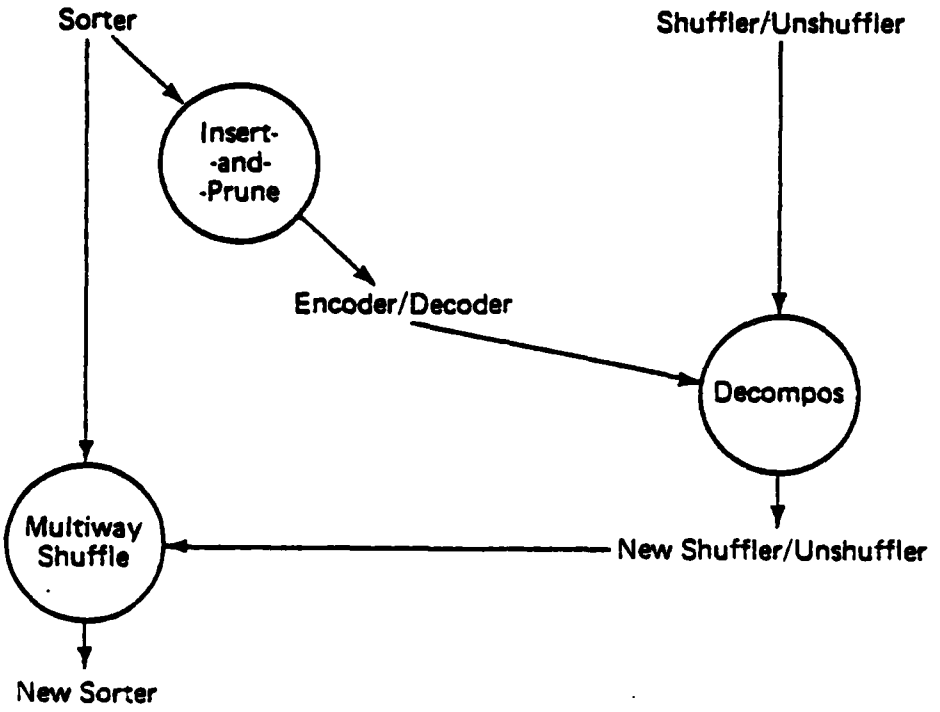


Figure 7.1. Basic steps in the construction of sorters for keys of medium length.

In Sections 7.2.1 and 7.2.2 we shall describe in detail the basic steps of Figure 7.1. Indeed one of them, which is the multiway-shuffle combination sort, has already been discussed in Section 5.2.4.

The ideas we have introduced could be applied to obtain boundary-chip sorters as well as non-boundary-chip sorters, although the construction of the latter is somewhat more involved. For the sake of simplicity, we shall develop the boundary-chip case, and we shall adopt the following conventions. All our circuits for sorting, encoding/decoding, and shuffling/unshuffling will be laid out in a region of rectangular shape, with the input ports on the north side and the output ports on the south side of the rectangle. The width of the rectangle will then be proportional to the number of I/O bits divided by computation time. The height will instead depend on the bisection flow that we are able to achieve.

and will be our objective limitation (to be reduced). Thus, for a complete $(n, \log n + h)$-sorter, our target is a width $O(n \log n / T^2)$ and a height $O(nh / T^2)$.

7.2.1 Insert-and-Prune Encoder and Decoder

We recall from Section 2.2.4 that the insert-and-prune encoding of a multiset $\{X_0, \ldots, X_{n-1}\}$ of words of length $\log n + h$ is obtained by sorting the multiset

$$\{X_0, \ldots, X_{n-1}\} \cup \{2^h i : i = 0, \ldots, n-1\},$$

and pruning the $\log n - 1$ most significant bits of each word in the resulting sequence.

Thus, an insert-and-prune encoder can be easily realized by a simple modification of any of the sorters described in Chapter 6. Indeed, it is sufficient to consider a $(2n, \log n + h)$-sorter such that n of the input keys are prestored and have the fixed values $0, 2^h, 2 \times 2^h, \ldots, (n-1)2^h$. The performance of such encoder is then $AT^2 = O(n^2 \log^2 n)$ for $T \in [\Omega(\log n), O(\sqrt{n \log n})]$.

The decoder is slightly more complex. Let the insert-and-prune encoding of $\{X_0, \ldots, X_{n-1}\}$ consists of the sequence of $2n$ words $(W_0, W_1, \ldots, W_{2n-1})$ of $h + 1$ bits each, with $W_i = W_i^h W_i^{h-1} \cdots W_i^0$. The following algorithm enables us to obtain the X's from the W's.

1. For $i = 0, 1, \ldots, 2n-1$, compute the value of the binary variable b_i defined as:

   $b_i = 0$ if either $(i = 0)$ or $(i > 0$ and $W_i^h = W_{i-1}^h)$

   $b_i = 1$ if $(i > 0$ and $W_i^h \neq W_{i-1}^h)$.

2. Compute the cumulative sum of the sequence b_i, defined as $B_i = \sum_{h=0}^{i} b_h$, where $B_i = B_i^{\log n - 1} B_i^{\log n - 2} \cdots B_i^1 B_i^0$ is a word of $\log n$ bits.

3. Form a new list $\overline{W}_0, \ldots, \overline{W}_{2n-1}$ where $\overline{W}_i = b_i B_i^{\log n - 1} \ldots B_i^1 W_i^h \ldots W_i^0$.

4. Sort the $\overline{W}_i$, and prune the most significant list of each key. The first n keys of the resulting sequence are $(Y_0, \ldots, Y_{n-1}) = sort\ (X_0, \ldots, X_{n-1})$, and they form the sorted list representation of the multiset encoded by $(W_0, \ldots, W_{2n-1})$.

Step 4 poses the heaviest demand of area-time resources. Thus, the insert- and-prune decoder can be also realized with $AT^2 = O(n^2 \log^2 n)$, for $T \in [\Omega(\log n), O(\sqrt{n \log n})]$.

In general, we can use the construction outlined above to obtain an encoder or a decoder from any given sorter. It is also convenient for our applications to combine the encoder and the decoder into one block, whose performance is stated in the following lemma.

Lemma 7.1. Given a design for an (n,k)-sorter with computation time $T_s(n,k)$ and height $H_s(n,k)$, we can construct an encoder/decoder with time and height respectively given by

$$T_{ED}(n,k) \leq \xi T_s(n,k) \tag{7.3}$$

and

$$H_{ED}(n,k) \leq \eta H_s(n,k), \tag{7.4}$$

where ξ and η are suitable constants (independent of n and k), greater than one.

7.2.2 Reducing the Bandwidth for Shuffling and Unshuffling

The multiway-shuffle combination (Section 5.2.4) is so denoted because two of the steps of the algorithm respectively consist of a p-unshuffle and of a p-shuffle of ml elements (where p divides l).

Indeed, these two steps are the only ones that require a global rearrangement of the input keys, and therefore pose the heaviest demand of bandwidth. Thus, it is crucial to be able to perform the shuffling and the unshuffling very efficiently.

In general, both the multiway shuffle and the multiway unshuffle of N words of K bits can each be executed by a circuit that works in time $T_{S\frac{1}{L}}$ and has width $O(NK/T_{S\frac{1}{L}})$ and height $H_{S\frac{1}{L}} \leq \sigma NKT_{S\frac{1}{L}} = O(NK/T_{S\frac{1}{L}})$, where σ is a constant. The lengthy, but rather straightforward details are not given here. (A network for similar operations is described in some detail in [BS 84].)

Although in the general case the performance of the circuit mentioned above is optimal, in the specific application we have in mind, the shuffle and unshuffle are performed on sequences that can be decomposed into sorted subsequences, which can be compressed by encoding techniques. As a result we

can achieve a smaller height for the circuit.

We shall exploit the following decomposition of the multiway-unshuffle and of the multiway-shuffle permutations, illustrated in Figure 7.2.

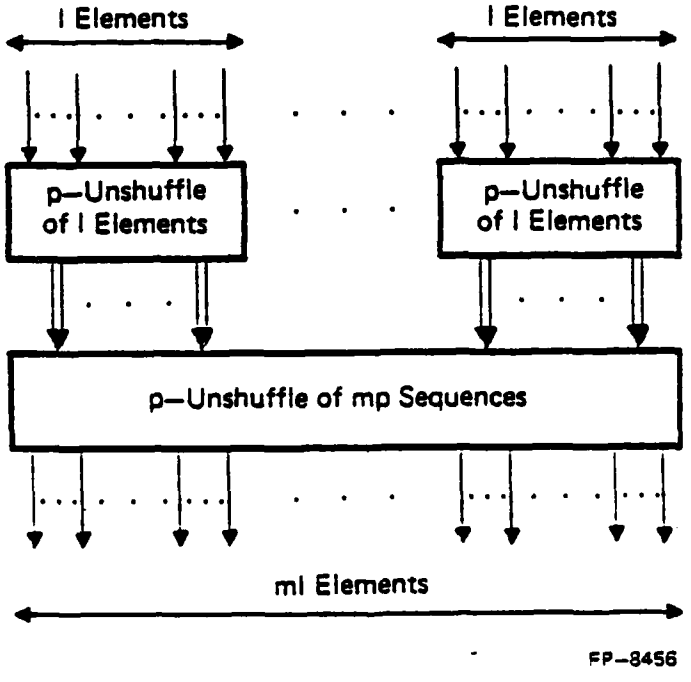


Figure 7.2. Cascade decomposition of the p-UNSHUFFLE of ml elements. (Arrows represent sequences of l/p elements.)

(1) The p-unshuffle of ml elements (where p divides l) can be performed by (1a) applying a p-unshuffle to each of the m subsequences that we can form with l consecutive elements, and then (1b) applying a p-unshuffle to the sequence of the mp sequences (regarded as single words) of l/p consecutive elements in the arrangement resulting from (1a).

(2) The p-shuffle of ml elements (where p divides l) can be performed by (2a) applying a p-shuffle to mp sequences (regarded as single words) of l/p consecutive elements, and then (2b) applying a p-shuffle to each of the m subsequences of l consecutive elements in the arrangement resulting from (2a).

We plan to use a shuffler/unshuffler block as part of a multiway-shuffle combiner. In this context, the sequence to be shuffled or unshuffled consists of m sorted subsequences of l consecutive elements each. In this case, it is easy to see that the sequences that are regarded as words in the second stage of the decomposition are sorted. Thus, they can be encoded by the insert-and-prune method, and then be recovered with appropriate decoding. This consideration suggests the scheme of Figure 7.3 for the entire unshuffle operation. A similar scheme works for the shuffle. Obviously the same method would not work for unsorted inputs, since after encoding we would be able to recover only the multiset underlying the encoded sequence, but not the sequence itself.

If in the design of Figure 7.3 we make the unshuffling blocks bidirectional, and we replace encoders and decoders with encoder/decoder blocks, we obtain a network that can also shuffle. We now analyze the performance of such shuffler/unshuffler block, for the case when $p = m$ and under the assumption that we use building blocks with the following features (for later convenience, we use a superscript i to denote quantities related to building blocks, and a superscript $(i+1)$ to denote quantities related to the overall design).

(a) The encoder/decoder blocks which operate on sequences of n/m^2 elements ($n \triangleq ml$) of k bits each, work in time $T^i_{ED}(n/m^2, k)$ and have height $H^i_{ED}(n/m^2, k)$.

(b) The shuffler/unshuffler blocks which operate on sequences of n/m elements of k bits each, work in time $T^i_{SU}(n/m, k)$, and have height $H^i_{SU}(n/m, k)$.

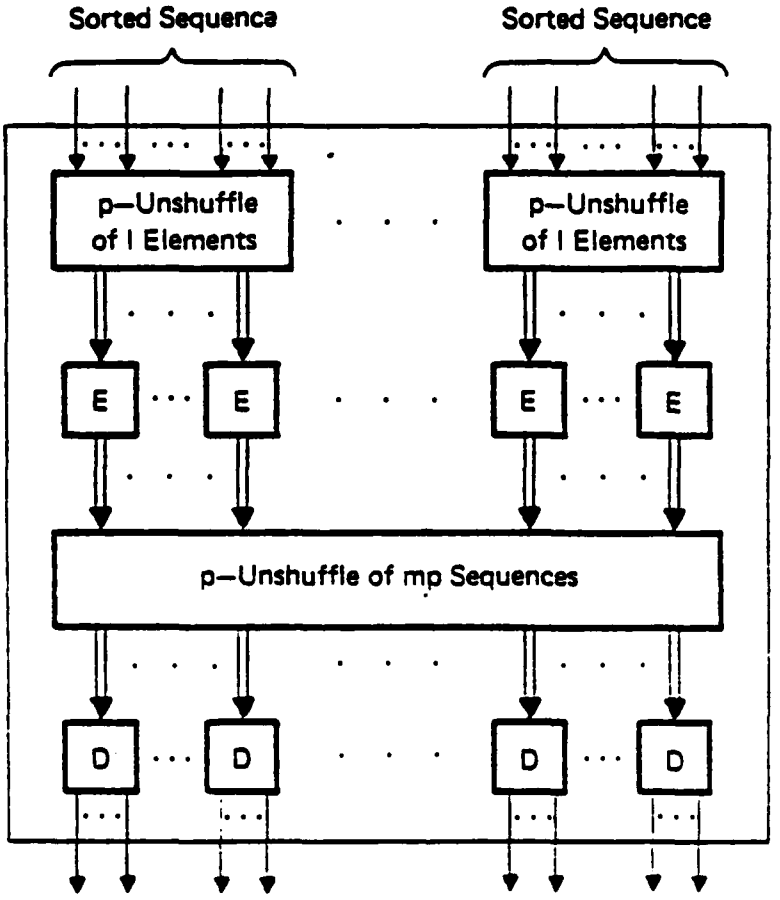


Figure 7.3. Cascade decomposition of the p-UNSHUFFLE of m elements when each of the m subsequences input by blocks of the first stage are sorted. Encoders (E) and decoders (D) operate on subsequences of l/p elements.

(c) The shuffler/unshuffler block, which operates on m^2 (encoded) sequences, is realized according to the straightforward method mentioned at the beginning of this section. Here $N = m^2$, since the items being shuffled are m^2, and each item consists of a sequence of n/m^2 words each represented

with $h_{i+1} \triangleq k - \log n + 2 \log m$ bits (see the insert-and-prune encoding). Thus,

$$H_{SU}^{1} \leq \sigma 2 n h_m / T_{SU}^{1}. \qquad 7.5$$

It is then easy to see that the performance of the entire shuffler/unshuffler circuits obtained by cascading the different stages is given by

$$H_{SU}^{i+1}(n,k) = 2\sigma n h_{i+1}/T_{SU}^{1} + H_{SU}^{i}(n/m,k) + 2H_{ED}^{i}(n/m^2,k) \qquad 7.6$$

$$T_{SU}^{i+1}(n,k) = T_{SU}^{1} + T_{SU}^{i}(n/m,k) + 2\,T_{ED}^{i}(n/m,k). \qquad 7.7$$

7.2.3 The Sorters

We consider now a network consisting of a set of m $(n/m,k)$-sorters and of an m-shuffler/unshuffler of n keys. Such a network can easily perform all the steps required by the multiway-shuffle combination algorithm described in Section 5. Obviously, the m sorters can also prepare the sorted sequences to be processed by the combiner, and - with small adaptations - they can also perform the sorting operation in the "windows" (refer to Section 5.2.4).

Thus, if we realize the sorters with a design with performance $T_S^i(n/m,k)$, $H_S^i(n/m,k)$, and the shuffler/unshuffler with a design with performance $H_{SU}^{i+1}(n,k)$, $T_{SU}^{i+1}(n,k)$, we obtain a sorter with global performance given by the following relations:

$$H_S^{i+1}(n,k) \leq H_{SU}^{i+1}(n,k) + H_S^{i}(n/m,k) \qquad 7.8$$

$$T_S^{i+1}(n,k) \leq \gamma_1 T_{SU}^{i+1}(n,k) + \gamma_2 T_S^{i}(n/m,k) \qquad 7.9$$

where γ_1 and γ_2 are constants. In fact the $(n/m,k)$-sorters, and the shuffler/unshuffler are activated a constant number of times during the entire algorithm.

With reference to Figure 7.1, we have now completed the description of the steps that allow to obtain the "new shuffler/unshuffler" and the "new sorter", given a sorter and a shuffler/unshuffler. We shall repeatedly use these steps to construct a sequence of designs, as follows.

We begin with a design for the sorter having performance

$$H_S^1(n,k)\,T_S^1(n,k) \leq C_S^1 nk \tag{7.10}$$

where C_S^1 is a constant. Such performance can be achieved by any of the sorters described in Chapter 6, as long as

$$\tau_S^1 \log n \leq T_S^1(n,k) \leq \hat{\tau}_S^1 \sqrt{n \log n} \tag{7.11}$$

for suitable constants τ_S^1 and $\hat{\tau}_S^1$. For the shuffler/unshuffler we begin with the straightforward implementation that achieves

$$H_{SU}^1(n,k)\,T_{SU}^1(n,k) \leq \sigma nk \tag{7.12}$$

as long as

$$\tau_{SU}^1 \leq T_{SU}^1(n,k) \leq \hat{\tau}_{SU}^1 \sqrt{nk} \tag{7.13}$$

for suitable constants τ_{SU}^1 and $\hat{\tau}_{SU}^1$.

We can then define a sequence of designs where the $(i+1)$-th one is obtained from the i-th one according to the scheme illustrated in Figure 7.1. A value m_i must also be chosen for the parameter m specifying how many sequences of n/m keys are to be presorted by sorters of the i-th type. We shall choose $m_i = L_i(n)$ where L_i is the i-th iterate of the logarithm, formally defined by

$$L_1(n) \triangleq \log n \tag{7.14}$$

$$L_i(n) \triangleq \log(L_{i-1}(n)), \qquad i > 1 \tag{7.15}$$

We claim that the sequence of sorters and shuffler/unshufflers so defined satisfies the relations

$$H_S^i T_S^i \leq C_S^i n h_i \tag{7.16}$$

$$H_{SU}^i T_{SU}^i \leq C_{SU}^i n h_i \tag{7.17}$$

for n large enough, when C_S^i and C_{SU}^i are constants, and

$$h_i \triangleq k - \log n + 2L_i(n). \tag{7.18}$$

Inequalities 7.16 and 7.17 can be proved by induction. For $i = 1$, they follow from 7.10 and 7.12 with any C_{SU}^1 greater than or equal to σ.

In general, using 7.3, 7.4, 7.8, 7.9, 7.10 and 7.12, substituting $L_i(m)$ for m, and taking 7.16 and 7.17 as inductive hypotheses, we obtain

$$H_{ED}^i T_{ED}^i \leq \eta \xi C_S^i n h_i \tag{7.19}$$

$$\begin{aligned} H_{SU}^{i+1} \leq\ & 2\sigma n h_{i+1} / T_{SU}^1 \\ & + C_{SU}^i n h_i / (L_i(n) T_{SU}^i(n / L_i(n), k)) \\ & + \eta \xi C_S^i n h_i / (L_i^2(n) T_S^i(n / L_i^2(n), k)) \end{aligned} \tag{7.20}$$

$$T_{SU}^{i+1} \leq T_{SU}^1 + T_{SU}^i(n / L_i(n), k) + 2T_S^i(n / L_i^2(n), k) \tag{7.21}$$

$$H_S^{i+1} \leq H_{SU}^{i+1} + H_S^i(n / L_i(n), k) \tag{7.22}$$

$$T_S^{i+1} \leq \gamma_1 T_{SU}^{i+1} + \gamma_2 T_S^i(n / L_i(n), k). \tag{7.23}$$

We are further allowed to choose $T_S^i(n / L_i^2(n), k)$ and $T_{SU}^i(n / L_i(n), k)$ within the range of possible sorting and shuffling/unshuffling computation times relative to the i-th design. If we choose them to be proportional to T_{SU}^i, then inequalities 7.20 and 7.21 imply that

$$H_{SU}^{i+1} T_{SU}^{i+1} \leq 2\alpha\sigma n h_{i+1} + o(n h_i / L_i(n)). \tag{7.24}$$

If $k = \log n + O(L_i(n))$, then $h_i / L_i(n) = O(1)$, and we obtain

$$H_{SU}^{i+1} T_{SU}^{i+1} \leq 2\alpha\sigma n h_{i+1} + \text{lower order terms} \tag{7.25}$$

Under the same assumptions, inequalities 7.24 and 7.25 yield for the sorter

$$H_S^{i+1} T_S^{i+1} \leq 2\alpha\sigma\gamma_1 n h_{i+1} + \text{lower order terms}. \tag{7.26}$$

Thus, if we define

$$C_S^i \triangleq 2\alpha\sigma\gamma_1, \tag{7.27}$$

$$C_{SL}^i \triangleq 2\alpha\sigma, \tag{7.28}$$

then 7.26 and 7.25 show that inequalities 7.16 and 7.17 hold for $i + 1$.

The previous discussion can be summarized as in the following theorem.

Theorem 7.1 For any $i \geq 1$, an $(n, \log n + O(L_i(n)))$-sorter with I/O ports on the boundary can be constructed, such that

$$AT^2 = O(n^2 \log n L_i(n)) \tag{7.29}$$

for $T \in [\Omega(\log n), O(\sqrt{nL_i(n)})]$. Such sorter is optimal if $k = \log n + \theta(L_i(n))$.

The ideas exploited in this section could also be used to design non-boundary $(n, \log n + O(L_i(n)))$-sorters with $AT^2 = O(n^2 L_i^2(n))$. However the constructions are rather elaborate and do not add further insight to the problem of sorting medium words, and therefore are not reported here.

7.3 SORTERS FOR SHORT KEYS

In this section we derive upper bounds for the (n,k)-sorting problem when the keys are short, i.e. when $k \leq \log n$, or equivalently, when the size $z = 2^k$ of the universe is not larger than the size n of the multiset being sorted.

We recall from Chapter 4 the lower bounds for this problem. We restrict our attention to word-local designs. In fact, as indicated by Theorem 4.8, non-word-local protocols lead to larger information exchange than word-local ones.

It is useful to introduce the quantity

$$d \triangleq n/r \tag{7.30}$$

which, as we shall see below, plays an important role. Then, for boundary chips, we have from Theorem 4.7 that

$$AT^2 = \Omega(d \ log n \ r^2 \log r) \qquad 7.31$$

For non-boundary chips, Theorems 4.9 and 4.10 respectively yield

$$AT^2 = \Omega(d \ r^2) \qquad 7.32$$

and

$$AT = \Omega(d \ r^{3/2}) \qquad 7.33$$

Furthermore, Theorems 4.11 and 4.12 tell us that

$$A = \Omega(r \ \log(l + n/r)) \qquad 7.34$$

and

$$T = \Omega(log n). \qquad 7.35$$

7.3.1 The Algorithm

Here we propose a new sorting algirithm, specifically tailored to short keys, and we also describe a VLSI implementation of it, whose performance comes very close to the above lower bounds.

The main idea of the algorithm consists in using an efficient encoding for multisets of small keys in the intermediate stages of the sorting process. We shall in fact encode a multiset S by means of its distribution function. Let us recall (Equation 2.15) that if S is a multiset on the universe $U = \{0,1,\ldots,r-1\}$, then the multiplicity of an element $i \in U$ is defined as

$$\mu(i) \triangleq number \ of \ occurrences \ of \ element \ i \ in \ multiset \ S,$$

and the distribution function (Equation 2.19) is defined as the vector

$$(M(i) \triangleq \sum_{j \le i} \mu(j): \ i = 0,1,\ldots,r-1).$$

A simple but useful property is that the distribution of the union of two multisets S and R is simply

$$M_{S \cup R}(i) = M_S(i) + M_R(i), \qquad 7.36$$

with obvious meaning of the symbols. Thus, we can say that the merging of two sequences is

transformed - in the distribution encoding - into the sum of their distribution functions. This property is used to design the following simple algorithm:

1. (ENCODE) Subdivide the input multiset $\{X_0, \ldots, X_{n-1}\}$ into $d = n/r$ submultisets of r keys each, and compute the distribution function of each submultiset.
2. (TALLY) Sum the d distribution functions (as r-component vectors) obtained in Step 1, to produce the (global) distribution function of the entire input multiset.
3. (BROADCAST) Replicate the global distribution function d times.
4. (DECODE) From the i-th replica of the distribution function obtain the r consecutive output keys $Y_{ir}, Y_{ir+1}, \ldots, Y_{ir+r-1}$ $(i = 0,1,\ldots,d-1)$, with a suitable decoding procedure.

The rationale for Step 4 is the wish to deploy decoders comparable to the corresponding encoders; this creates the need for Step 3, the d-way replication of the distribution vector.

A preliminary step is the discussion of the algorithms for encoding and decoding, which turn out to be based on merging and sorting operations.

7.3.2. Transcoding Operations

In order for the algorithm outlined above to be efficient, we need an efficient way to obtain the distribution encoding of a multiset from its list representation, and vice versa. We propose now some algorithms to perform these transformations of encodings.

List-to-Distribution (Encoding). Given a multiset S represented by a list $\{X_0, X_1, \ldots, X_{n-1}\}$ with $X_i \in U = \{0,1,\ldots,r-1\}$, we define a sorted list

$$Z = (Z_0, Z_1, \ldots, Z_{n+r-1}) \triangleq sort(S \cup U). \tag{7.37}$$

If $\mu(i)$ is the multiplicity of i in S, then the structure of Z is a concatenation of runs of identical symbols

$$Z = (\underbrace{0,\ldots,0}_{\mu(0)+1}, \underbrace{1,\ldots,1}_{\mu(1)+1}, \ldots, \underbrace{r-1,\ldots,r-1}_{\mu(r-1)+1}) \tag{7.38}$$

If we consider the last element in a run of the form $i,\ldots,\underline{i}$ (underscored in 7.38), we can see that its index in the sequence Z is $b = M(i) + i$, where $M(i) = \mu(0) + \mu(1) + \ldots + \mu(i)$. The last element in a run can be easily recognized because it differs from its successor. Thus, we can construct a sequence W' defined as

$$W'_b = \begin{cases} b - Z_b & \text{if } Z_{b+1} \neq Z_b \quad (\textit{that is}, W'_b = M(Z_b)) \\ n & \text{if } Z_{b+1} = Z_b \end{cases} \qquad 7.39$$

If we sort W' and define $W \triangleq sort\ (W')$, all the elements of W' that are equal to n will occupy the last n position of W, and we can extract the distribution of M of multiset S from the first r positions, i.e.:

$$M = (M(0),\ldots,M(r-1)) = (W_0,\ldots,W_{r-1}) \qquad 7.40$$

If necessary, the multiplicity could be obtained as $\mu(i) = M(i) - M(i-1)$, where $M(-1) \triangleq 0$.

Example.

$S = \{0,1,1,2,4,4,4,6,7,7\}$, $(n = 10)$, $U = \{0,1,2,3,4,5,6,7\}$, $(r = 8)$.

$Z = (0,\underline{0},1,1,\underline{1},2,\underline{2},\underline{3},4,4,4,\underline{4},\underline{5},6,\underline{6},7,7,\underline{7})$

$W' = (10,\underline{1},10,10,\underline{3},10,\underline{4},\underline{4},10,10,10,\underline{7},10,\underline{8},10,10,\underline{10})$

$W = (1,3,4,4,7,7,8,10,10,\ldots,10)$

$M = (1,3,4,4,7,7,8,10)$

$\mu = (1,2,1,0,3,0,1,2)$.

Distribution-to-list (decoding). Given the distribution vector $M = (M(0),M(1),\ldots,M(r-1))$ of a multiset S, whose sorted list representation is $(Y_0,Y_1,\ldots Y_{n-1})$, we want to compute a set of p consecutive elements of this list starting at Y_b, i.e. we want to compute $(Y_b,Y_{b+1},\ldots Y_{b+p-1})$. Obviously, if $b = 0$ and $p = n$, we obtain the entire sorted sequence of S. However, as we shall see, it is useful to be able to compute different portions of sequence $(Y_0,\ldots Y_{n-1})$ independently of each other.

The method proposed is based on the following idea. If $Y_h = i$, then there are at least $h - 1$ elements of S not larger than i, and at most h elements smaller than i, so that $M(i-1) \leq h < M(i)$. Thus, if we insert h $(0 \leq h \leq n-1)$ into the sorted sequence $M = (M(0),\ldots,M(r-1))$, and find the

value i such that $M(i-1) \leqslant h < M(i)$ we can conclude that $Y_h = i$.

Then, if we want to compute $Y_b, Y_{b+1}, \ldots, Y_{b+p-1}$, we have to simultaneously insert the elements of the sequence $B = (b, b+1, \ldots, b+p-1)$ into sequence M, which can be done by merging M and B. For later use, we first append to each of the keys to be merged a tag field with value zero for elements in M, and with value one for elements in B. Then we merge M and B in a stable way obtaining a sequence

$$W = merge\ (M, B). \qquad 7.41$$

In sequence W, an element h of B follows $M(0), M(1), \ldots, M(Y_h - 1)$, as well as $b, b+1, \ldots, h-1$, and will therefore occupy the $(Y_h + h - b)$-th position. If the q-th element of W comes from sequence B (which we can test from the tag field) and has value h, then we update it as

$$W_q = q - (W_q - b) = Y_h. \qquad 7.42$$

At this point, by a simple unmerge of the sequences of zero tags and one tags, we obtain a sequence of one-tag elements equal to $(Y_b, Y_{b+1}, \ldots, Y_{b+p-1})$.

Example. $M = (1,3,4,4,7,7,8,10)$. (Multiset S is the same as in the previous example.) $B = (6,7,8)$, i.e. $b = 6$, and $p = 3$. If we denote "tag-one" by underscoring we have

$$W = (1,2,4,4,\underline{6},7,\underline{7},8,\underline{8},10).$$

The three underscored elements W_4, W_7, and W_9 are updated according to 7.42 yielding $W_4 = 4 - (6-6) = 4$, $W_7 = 7 - (7-6) = 6$, $W_9 = 9 - (8-6) = 7$, so that $Y_6 = 4, Y_7 = 6$ and $Y_8 = 7$.

In general, the elements of both $M = (M(0), \ldots, M(r-1))$ and $B = (b, b+1, \ldots, b+p-1)$ are numbers in the range $0,1, \ldots, n$-1, and their binary representation requires $\log n$ bits. We discuss now some modifications of the above procedure that allow us to work with numbers with k+1 bits, at least in the case when $b = ir$, and $p = r$, which is needed in our sorting algorithm.

If $B = (ir, ir+1, \ldots, ir+ir-1)$, we can replace $M(h)$ by ir whenever $M(h) < ir$, and by $(i+1)r$ whenever $M(h) > (i+1)r$, without affecting the order of elements of sequences B and M. This observation suggests the definition of a new sequence, which we call the i-modified distribution function, i.e.

$$\overline{M}_i(h) = \begin{cases} ir & \text{if } M(h) < ir \\ M(h) & \text{if } ir \leq M(h) \leq (i+1)r \\ (i+1)r & \text{if } M(h) > (i+1)r \end{cases} \qquad 7.43$$

Then, the sequence $(Y_{ir}, Y_{ir+1}, \ldots, Y_{ir+r-1})$ can be obtained by a straightforward modification of the above decoding procedure, operating on the squences $\overline{M}_i = (\overline{M}_i(0), \ldots, \overline{M}_i(r-1))$, and $\overline{B} = (0,1,\ldots,r-1)$, rather than on sequences M and B. The advantage lies in the fact that elements of $\overline{M}_i$ and $\overline{B}$ can be represented with $k+1$ bit (instead of $\log n$).

In the sorting algorithm outlined in Section 7.3.1, both the encoding and the decoding procedures are applied to multisets of r elements of $k = \log r$ bits each. It is then easy to see that both the encoder and the decoder can be realized as simple modifications of a $(2r, \log r+1)$ sorter. These modifications can be done without affecting the (order of the) area-time performance of the sorter itself.

7.3.3 The Network

We discuss first a nonpipelined version of the network, and then we obtain the area-time trade-off by means of a pipelined version.

We recall that n, $r = 2^k$, and $d = n/r$ are powers of two, and we introduce the following subsequences of the input and of the output sequences of the sorter:

$$S_i \triangleq (X_{ir}, X_{ir+1}, \ldots, X_{ir+r-1}),$$

$$R_i \triangleq (Y_{ir}, Y_{ir+1}, \ldots, Y_{ir+r-1}).$$

We also consider the distribution function of multiset S_i.

$$M_i = (M_i(0), \ldots, M_i(r-1))$$

which is a vector with r $(k+1)$-bit components.

The nonpipelined version of the sorting network is the cascade of four parts, illustrated in Figure 7.4, each performing one of the four steps of the algorithm.

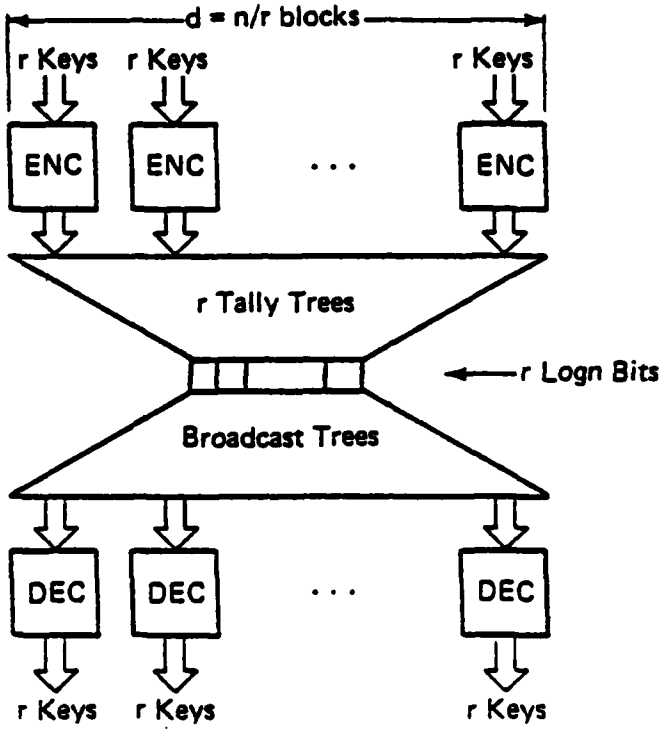


Figure 7.4. Structure of the network for sorting a set of short keys.

(a) (ENCODERS) Encoders $E_0, \ldots, E_{d-1}$, each capable of computing the distribution of a given (r,k)-multiset. Encoder E_j inputs S_j and computes M_j. We assume that each encoder has r input lines and r output lines, and that I/O operations on words are bit-serial.

(b) (TALLY TREES) Tally trees $TL_0, \ldots, TL_{r-1}$, each a full binary tree on d leaves, where a node at distance l from the leaves is equipped with an $O(l)$-bit storage and an l-bit operand carry-save adder, and is connected to its father by $O(l)$ wires. The j-th leaf of tally tree TL_h is connected to the h-th output line of encoder E_j, from which it will read - in bit serial fashion, LSB first - the distribution value $M_j(h)$. By summing $M_0(h), \ldots, M_{d-1}(h)$, TL_h computes $M(h)$. Thus each tree tallies d k-bit numbers to produce a $(k+\log d) = \log n$ bit result. The operation of a tally tree

is illustrated in Figure 7.5. First, for each bit position we obtain a log$d$-bit count of its 1's (this is done by suitable adders at the nodes of the tree); next the bit-counts are added with the correct alignment (carry-release) at the root of the tree. Each of these additions is performed in $O(1)$ time on a redundant carry-save representation. The conversion from carry-save to standard is done at the end of the step in time $O(k)$. Note that at any time *only one level* of the tree is occupied by data generated by a given bit position.

(c) (BROADCAST TREES) Broadcast trees $BC_0, \ldots, BC_{r-1}$, are similar in structure to the tally trees, but different in the functional capabilities of their nodes. The h-th leaf of broadcast tree BC_j is connected to the h-th input line of decoder D_j, to which the value $\bar{M}_j(h) \bmod r$ must be transmitted. Let j_0 be such that $j_0 r \leq M(h) \leq (j_0+1)r$. Then leaves $0,1,\ldots,j_0-1$ of BC_h must receive the value r, leaf j_0 receives $M(h) \bmod r$, and leaves $j_0+1,\ldots,d-1$ must receive the value 0. This is done as follows. The log$d$ = log$n$-k most significant bits of $M(h)$, which are indeed the binary expansion of j_0, are used to set leaf j_0 to receive the k least significant bits of $M(h)$ and to appropriately force all other leaves. This would be trivial if log$n$ time were allowed

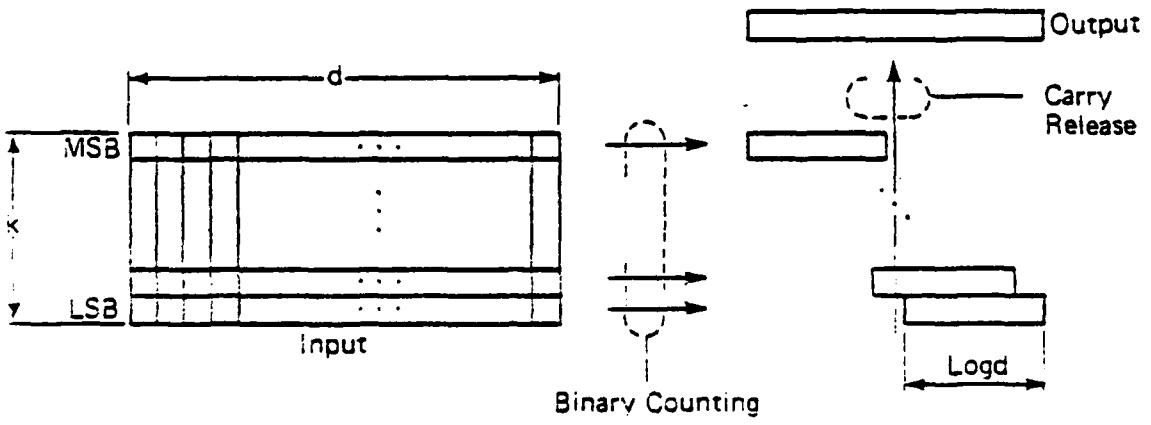


Figure 7.5. Tally-tree function.

for this operation. However, since we only allow time k,the $\log d$ MS-bits are injected in parallel into the root, and trace the path to leaf j_0 losing one (most significant) bit at each level; the least significant k bits follow serially.

(d) (DECODERS) Decoders $D_0,\ldots,D_{d-1}$, each capable of computing the portion R_i of the output sequence from the appropriate modified distribution of the entire input multiset. The I/O operations are performed with a protocol similar to the one used by the encoders.

An important remark is that the above network has period k. Therefore it can be used in a pipeline fashion with this period. This leads to the final sorting network. Letting $d = d_1 d_2$ and $r = r_1 r_2$ (since d and r are powers of 2, so are their factors), the network has d_2 encoders and d_2 decoders, each with r_2 input and output lines. Correspondingly, there are r_2 tally and broadcast trees, each with d_2 leaves. In this network, a given encoder will process d_1 different multisets (E_j will process $S_j, S_{j+d_2},\ldots,S_{j+d-d_2}$), and a given decoder will compute d_1 different subsequences of the output (D_j computes $R_j, R_{j+d_2},\ldots,R_{j+d-2}$). Each "wavefront" has a depth of k-bits, so that the period of the network matches the depth of each pipelined wavefront.

7.3.4 Area-Time Performance

We shall focus on encoders and tally trees, since decoders and broadcast trees are analogous.

An encoder with r_2 I/O lines can be realized as a modification of an $(r,\log r + \theta(\log r))$-sorter (see Chapter 6), with performance $A = O(r_2^2)$, $T = O(kr/r_2)$, for r_2 in the range $\sqrt{kr} \leq r_2 \leq r$.

The tally tree structure, with d_2 leaves and edge-bandwidth r_2, can be laid out in $O(d_2 r_2^2)$ area, by using the H-tree scheme. This area also accounts for the encoder modules.

Finally, adding the contribution of the r $\log n$-bit registers deployed to store the r values of the distribution, we obtain a global area

$$A = O(d_2 r_2^2 + r\log n) \qquad 7.44$$

where $1 \leq d_2 \leq d$.

The running time is of the form (for suitable constants C_1 and C_2):

$$T = C_1(Ckrd_1 + \log d_2). \qquad 7.45$$

In fact, an encoder spends $O(kr_1)$ time to process the r_1 data wavefronts for each of the d_1 subproblems assigned to it. A similar performance is achieved by the tally trees when used in pipeline, with the addition of the terms $\log d_2$ representing the depth of the pipe. Recalling that $r_1 = r/r_2$, $d_1 = d/d_2$, and $d = n/r$, 7.45 can be rewritten as

$$T = C_1(C_2kn/(r_2d_2) + \log d_2). \qquad 7.46$$

At this point, the analysis of the network performance is complete. However, we can still optimize the choice of r_2 and d_2. Formally, for each feasible value T of the computation time, we should minimize A (as given by 7.44) with respect to d_2 and r_2, which are subject to the appropriate constraints.

On an intuitive basis we expect the following facts. The minimum computation time should be achieved by the network with the maximum degree of parallelism, i.e. with maximum r_2 and d_2. To obtain slower networks we have two possibilities: one is to slow down the encoders and the decoders (by decreasing r_2), and the other is to decrease their number (d_2). As long as it is possible, we prefer to decrease r_2, because the area depends quadratically on r_2, and linearly on d_2 (see 7.44). However, when r_2 reaches its lower limit $\sqrt{kr}$, the only option left is decreasing d_2.

Thus, we shall obtain that for fast computations the area depends quadratically on $1/T$, and for slow computations the area depends linearly on $1/T$. This result is not surprising since we had already found a similar behavior for the lower bounds.

On a more quantitative basis we introduce the variable

$$r_2 \triangleq \min(r, C_2kn/(d\log d)) \qquad 7.47$$

and distinguish two cases:

1. $r_2^* > \sqrt{kr}$. In this case, if we let r_2 vary in the interval $[\sqrt{kr}, r_2^*]$ while keeping fixed $d_2 = d$, we obtain

$$AT^2 = O(d(kr)^2), \text{ for } T \in [\Omega(\log n), (\sqrt{kr})]. \quad 7.48$$

If we hold r_2 fixed and equal to $\sqrt{kr}$, and we let d_2 vary in the interval $[\log n/k, d]$, we obtain

$$AT = O(d(kr)^{3/2}), \text{ for } T \in (\Omega(\sqrt{kr}), O(k^{3/2}n/(\sqrt{r}\log n))]. \quad 7.49$$

For $d_2 < \log n/k$, then the term $r\log n$ prevails in the right-hand side of 7.43, so that no reduction in area would result by selecting a computation time larger than $\theta(k^{3/2}n/(\sqrt{r}\log n))$.

2. $r_2^* \leq \sqrt{kr}$. From 7.46 we can see that this condition is equivalent to $(1 + C_2 r)k \leq \log n$. In this case r is so small that even with the slowest encoder and decoder, the encoding/decoding time would be less than the tally/broadcast time, if we were to use d leaves in the tree structures. Thus, we define a value d_2^* by the equation

$$d_2^* \log d_2^* = C_1 \sqrt{kr}\, n/r, \quad 7.50$$

and we consider the class of networks obtained when $d \in [\log n/k, d_2^*]$ while $r_2 = \sqrt{rk}$. The performance is

$$AT = O(d(kr)^{3/2}), \text{ for } T \in [\Omega(\log n), O(k^{3/2}n/\sqrt{r}\log n)]. \quad 7.51$$

The above discussion is summarized by the following theorem.

Theorem 7.2. An (n,k)-sorter can be constructed, for $1 \leq k \leq \log n$, with the following performance ($r = 2^k$, $d = n/r$, C_2 a suitable constant).

$$AT^2 = O(d(kr)^2) \text{ for } T \in [\Omega(\log n)\, O(\sqrt{kr})], \quad 7.52$$

and

$$AT = O(d(kr)^{3/2}) \text{ for } T \in [\Omega(\sqrt{kr}), O(k^{3/2}n/(\sqrt{r}\log n))]. \quad 7.53$$

If $(1 + C_2 r)k \leq \log n$, then

$$AT = O(d(kr)^{3/2}) \text{ for } T \in [\Omega(\log n), O(k^{3/2}n/(\sqrt{r}\log n))]. \quad 7.54$$

Comparing the results of Theorem 7.2 with lower bounds 7.32 and 7.33, we can make the following

observations. In the range of computation times where the governing bounds are of the AT^2 form, there is an $O(k^2)$ gap. In the range where the governing bounds are of the AT form, the gap is instead $O(k^{3/2})$.

In fact, when manipulating multisets of r keys, in the form either of lists or of distributions, our circuits function on an $O(kr)$-bit representation of the multisets, where $O(r)$ bits are sufficient from an information-theoretic viewpoint.

One potentially useful modification is the use of sorter for medium-length keys, since encoders and decoders are based on $(2r,\log(2r))$-sorters. However, this would create a new problem, that is: once we keep the multisets of size r encoded with $O(r)$ bits, it is not immediate to see how the multiplicity of different multisets can be tallied.

Remark. The network described in Section 7.3.3 can also be laid out with all the I/O ports on the boundary. A simple analysis would show

$$A = O(d_2 \log^2 d_2 r_2^2 + r\log n), \tag{7.55}$$

and a result analogous to Theorem 7.2 can be obtained.

7.4 SORTERS FOR LONG KEYS

In this section we derive upper bounds for the (n,k)-sorting problem when the keys are long, i.e. when $k \geq 2\log n$.

We summarize first what we already know about the problem. The case of word-local protocols is easily taken care of. In fact, it is not difficult to realize that all constructions proposed in Chapter 6 achieve $AT^2 = O(k^2 n^2)$ on keys of arbitrary length k, thus attaining the $AT^2 = \Omega(k^2 n^2)$ lower bound of Theorem 4.13.

Thus, we turn our attention to non-word-local protocols. It is useful to define the quantity

$$d \triangleq k \log n, \tag{7.56}$$

which, as we shall soon see, plays an interesting role in the sorting of long words. Considering that $k = d\log n$, the lower bounds obtained in Theorems 4.18 and 4.19 can be respectively restated as

$$AT^2 = \Omega(d(n\log n)^2) \tag{7.57}$$

and

$$AT = \Omega(d(n\log n)^{3/2}). \tag{7.58}$$

Furthermore, Theorems 4.20 and 4.21 tell us that

$$A = \Omega(n\log n) \tag{7.59}$$

regardless of d (or k), and

$$T = \Omega(\log n + \log k) = \Omega(\log n + \log d). \tag{7.60}$$

The performance of known constructions discussed in Chapter 6 is

$$AT^2 = O(k^2 n^2) = O(d^2(n\log n)^2) \tag{7.61}$$

as we have mentioned above. Comparing bounds 7.57 and 7.61 we see that there is an $O(d)$ gap so that the known designs are optimal only if $d = O(1)$. The general case, when d increases with n, needs further investigation.

We shall present a new design of an (n,k)-sorter whose performance comes very close to the lower bounds 7.57 and 7.58.

7.4.1 A Non Word-Local Sorting Algorithm

From the preceding discussion, it is obvious that to improve the $AT^2 = O(d^2(n\log n)^2)$ upper bound we have to resort to non-word local algorithms. Moreover, the form of the lower bounds, which are linear in d, suggests the decomposition of the problem in d subproblems, whose solutions are combined with small information exchange.

The approach that we shall follow consists of decomposing the keys in blocks of consecutive bits, and then processing together the homologous blocks of different keys. A similar approach has been

considered by Leighton.[1] Some notation will be useful for our discussion. (Refer to Figure 7.6.)

For simplicity we assume that $n = 2^{\nu}$ (so that $\log n = \nu$ is an integer) and that $k = d \log n = d\,\nu$, for integer d. We observe that to require that $k/\log n$ is an integer is not a serious constraint, since we can always comply with it by adding less than $\log n$ bit positions to the keys without changing the input size significantly.

With the above assumptions, we can partition each key into d blocks of consecutive bits. We denote the h-th (least significant) block of key X_i by

$$X_i(h) = X_i^{(h+1)\nu-1} \ldots X_i^{h\nu}, \qquad 7.62$$

for $h = 0,1,\ldots,d-1$. (See Figure 7.6). A similar partition can be also considered for the output keys, defining

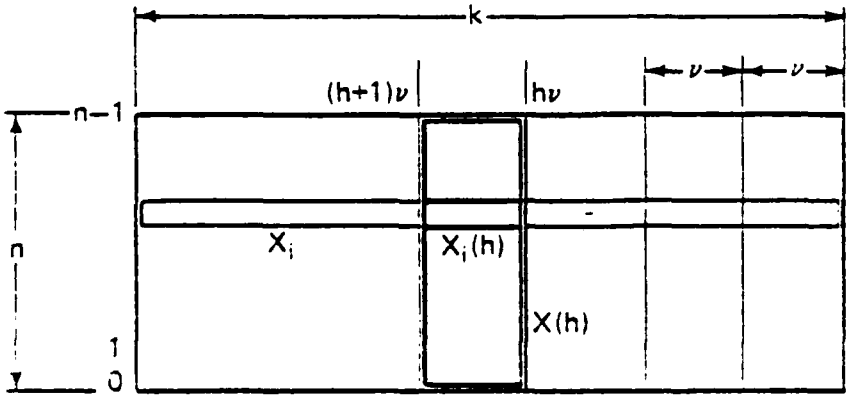


Figure 7.6. Nomenclature for input data.

[1]Personal communication.

$$Y_i(h) = Y_i^{(h+1)\nu-1} \ldots Y_i^{h\nu}. \qquad 7.63$$

It is obvious that, for given h, $(Y_0(h),\ldots,Y_{n-1}(h))$ is a permutation of $(X_0(h),\ldots,X_{n-1}(h))$. This permutation is functionally dependent on the values of the bits in blocks h, $h+1$, ... $d-1$, as we have already seen in Section 4.2, where the information transfer caused by this dependence has been informally called "secondary flow".

The rank of key X_i in the multiset $\{X_0,\ldots,X_{n-1}\}$ is the number of keys in the multiset that are strictly smaller than X_i. Formally

$$rank\ (X_i) = |\{j : X_j < X_i\}|. \qquad 7.64$$

The following property of the rank is very useful for us. Suppose that each key $X_0,\ldots,X_{n-1}$ is viewed as the concatenation of three strings, namely

$$X_i = L_i * C_i * R_i$$

where $*$ denotes string concatenation. Moreover the number of bits of L_i is not a function of i, and similarly for C_i and R_i. Then if we define

$$rank\ (C_i) = |\{j : C_j < C_i\}| \qquad 7.65$$

and we view $rank\ (C_i)$ as a binary string of ν bits, we have that

$$rank\ (X_i) = rank\ (L_i * rank\ (C_i) * R_i). \qquad 7.66$$

Equation 7.66 follows from the fact that $X_j < X_i$ if and only if $L_j * rank\ (C_j) * R_j < L_i * rank\ (C_i) * R_i$, whose proof is almost immediate.

If we consider the decomposition

$$X_i = X_i(d-1) * \ldots * X_i(h) * \ldots * X_i(0)$$

of the input keys, and we repeatedly apply Equation 7.66 we obtain

$$rank\ (X_i) = rank\ (rank\ (X_i(d-1)) * \ldots * rank\ (X_i(h)) * \ldots * rank\ (X_i(0))), \qquad 7.67$$

which, in words, says that the rank of the concatenation is the rank of the concatenation of the ranks.

This property allows us to reduce the computation of the rank of long keys to the computation of the rank of small substrings of the keys themselves, but we do not know yet how to compute the rank of the substrings. This problem can be solved by the following procedure, which is based on sorting.

(i) (EXTEND) To compute the ranks of the elements of $\{X_0, X_1, \ldots, X_{n-1}\}$ form a new set of keys $\overline{X}_i = X_i * i$, $i = 0, \ldots, n-1$ where i is represented with ν bits.

(ii) (SORT) Sort $\{\overline{X}_0, \ldots, \overline{X}_{n-1}\}$ to obtain a sorted sequence $\overline{Y}_0, \ldots \overline{Y}_{n-1}$. Then

$\overline{Y}_i = X_{\pi(i)} * \pi(i)$, $i = 0, \ldots, n-1$
where $\pi(0), \ldots, \pi(n-1)$ is a permutation of $0, \ldots, n-1$.

(iii) (RANK) Compute rank $(X_{\pi(i)})$ as one plus the maximum index j such that $X_{\pi(j)} < X_{\pi(i)}$. If no such index exists, then let rank $(X_{\pi(i)}) = 0$.

(iv) (EXTRACT) Form a new set of keys $Z_i = \pi(i) * rank(X_{\pi(i)}) * X_{\pi(i)}$ and sort $\{Z_0, \ldots, Z_{n-1}\}$, to obtain the sequence $(\hat{X}_0, \ldots, \hat{X}_{n-1})$ where $\hat{X}_i = i * rank(X_i) * X_i$.

Example. An example will illustrate the ranking algorithm. For simplicity we use digits instead of bits.

$(X_0, \ldots, X_6) = (7,6,1,4,4,7,9)$

$(X_0 * 0, \ldots, X_6 * 6) = (70,61,12,43,44,75,96)$

$(X_{\pi(0)} * \pi(0), \ldots, X_{\pi(6)} * \pi(6)) = (12,43,44,61,70,75,96)$

$(X_{\pi(0)}, \ldots, X_{\pi(6)}) = (1,4,4,6,7,7,9)$

$(rank(X_{\pi(0)}, \ldots, rank(X_{\pi(6)})) = (0,1,1,3,4,4,6)$.

Once the ranks have been computed they can be used to sort each of the blocks (into which the keys have been partitioned) independently of one another. Indeed, if

$V_i = rank(X_i) * X_i(h)$,

and $(W_0, \ldots, W_{n-1})$ is the sorted sequence corresponding to $\{V_0, \ldots, V_{n-1}\}$, then it is easy to see that

$W_i = rank(Y_i) * Y_i(h)$.

Summarizing the preceding discussion, we obtain the following divide-and-conquer sorting algorithm

1. (DIVIDE) Decompose the input keys $X_0, \ldots, X_{n-1}$ into d blocks of $\nu = \log n$ consecutive bits each, so that $X_i = X_i(d-1) * \ldots * X_i(h) * \ldots * X_i(0)$.

2. (SUBPROBLEMS) For each $h=0, \ldots, d-1$, compute $rank(X_0(h)), \ldots, rank(X_{n-1}(h))$ with respect to multiset $\{X_0(h), \ldots, X_{n-1}(h)\}$.

3. (MARRY) Compute the ranks of the X_i's using Equation 7.66. More specifically, with the simplifying assumption $d = 2^b$, we compute the right-hand side of 7.66 with a fully balanced tree of operations. Each operation has two input sequences and produces as output the sequence of the ranks of their concatenation.

4. (ROUTE) Replicate the sequence $(rank(X_0), \ldots, rank(X_{n-1}(h)))$ d times - one for each block - and sort the sequence $(rank(X_0) * X_0(h), \ldots, rank(X_{n-1}) * X_{n-1}(h))$ for $h = 0, 1, \ldots, d-1$, to obtain the seqence $(rank(Y_0) * Y_0(h), \ldots, rank(Y_{n-1}) * Y_{n-1}(h))$.

5. (OUTPUT) Obtain the output keys $Y_0, \ldots, Y_{n-1}$ as $Y_i = Y_i(d-1) * \ldots * \ldots * Y_i(0)$.

The algorithm we have just described has a shortcoming. In fact all the input keys must be read (step 1) in order to compute $rank(X_0), \ldots, rank(X_{n-1})$ (steps 2 and 3), and no data can be output until step 5. This shortcoming can be eliminated by modifying the algorithm according to the observation that to arrange in the correct order the bits of a given block it is sufficient to know the ranks of the blocks of greater significance.

We can proceed as follows. For simplicity, let $d = d_1 d_2$. Let us also denote by $X(h)$ the portion of the array X corresponding to the h-th block of the keys. (The rows of $X(h)$ are $X_0(h), \ldots, X_{n-1}(h)$.) Then, we organize $X(d-1), X(d-2), \ldots, X(0)$ in a $d_1 \times d_2$ array with the index of the block in row-major order (see Figure

7.7).

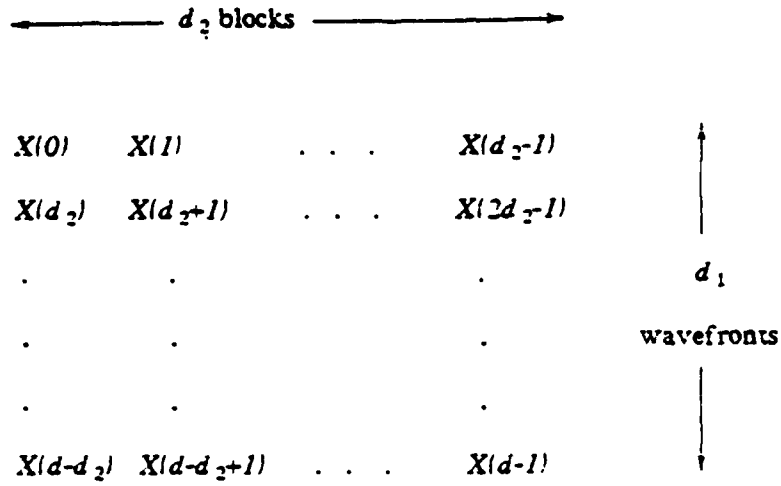


Figure 7.7. Organizing of the input for the pipelined sorting algorithm.

We propose a pipelined sorting algorithm with d_1 wavefronts of data, each of which is a row of the array just defined, and consists of d_2 blocks.

The topmost wavefront is processed exactly as described in the nonpipelined version of the algorithm. However, the ranks computed at step 3 are stored for later use. In fact for the second wavefront, once the ranks are computed they have to be further concatenated with the ranks of the first wavefront, and the ranks of the concatenation will drive the permutation of the data in the second wavefront. The computation proceeds in a similar fashion for the remaining wavefronts.

7.4.2 The Network

We now describe a network capable of executing the pipelined version of the sorting algorithm described above, with efficient area-time performance.

Figure 7.8 shows a high level representation of the network consisting of two tree structures and a family of linear arrays, whose interconnection and nodes are to be described in the following.

(a) *The ranking tree.* This component is a fully balanced binary tree on d_2 leaves, and d_2-1 internal nodes. Both the leaves and the internal nodes are essentially sorting modules with some further capabilities to compute the ranks of a sequence, although a leaf module performs a function

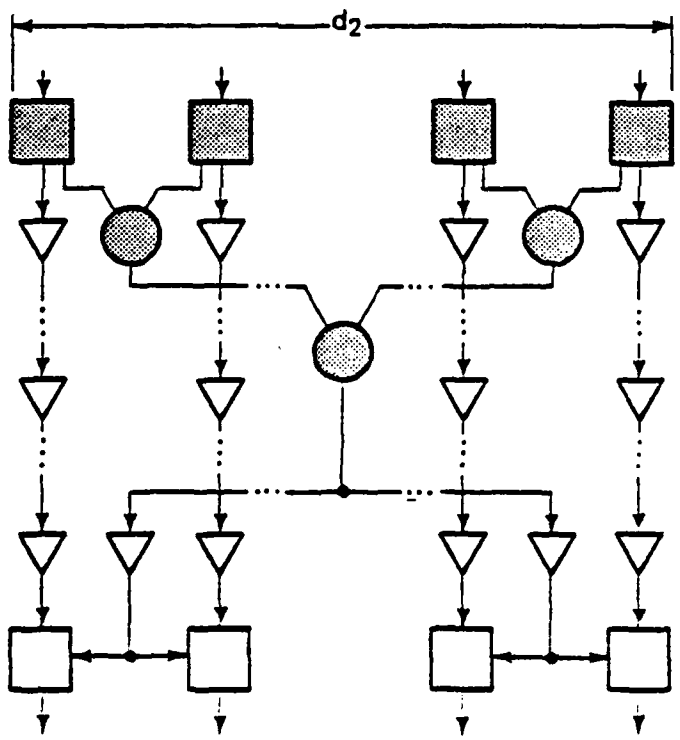


Figure 7.8. Structure of the network for sorting a set of long keys.

The above analysis is only approximate because it assumes that the area-time resources to combine the solutions to the subproblems are at most of the same order of those needed to obtain them. Indeed, the upper bounds of Chapter 7 are slightly larger than those suggested by the above analysis as well as by the square-tessellation lower bounds, indicating that further work is needed to complete the characterization of the area-time complexity of sorting.

Some interesting observations of general interest for VLSI computation can also be made by focussing on the architecture of the several kinds of sorters examined in this thesis. Although we have considered a variety of networks, all of them can be viewed as developments of three basic structures: the linear array, the binary tree, and the binary cube. Since this is true also of the networks used in VLSI designs proposed in the literature for the solution of other problems, there is a reasonable hope for a coherent and systematic theory of VLSI architectures.

The lower-bound, algorithmic, and architectural concepts introduced in this thesis will certainly be useful to analyze other problems related to sorting, like merging or sorting of records where there is another field beside the key. We also hope that they will contribute to the development of a comprehensive theory of VLSI computing.

appropriate architectures for the execution of classical algorithms for parallel sorting, such as bitonic sorting and merge-enumeration sorting.

The optimal sorters for $k = \log n + \theta(\log n)$ have been the basis for other circuits that can sort keys of arbitrary length. We have classified the keys as short ($k \leq \log n$), medium length ($\log n < k < 2\log n$), and long ($k \geq 2\log n$), and for each class we have proposed new algorithms which can be implemented with optimal or near-optimal area-time performance.

Of particular interest is the examination of the designs that sort short and long keys since the lower bounds on their performance is obtained by tessellation techniques. Indeed, these designs exhibit two different computational regimes, one for slow computation where the area of the circuits is (almost) proportional to $1/T$, and one for fast computation where the area of the circuits is proportional to $1/T^2$.

The presence of two different regimes can be explained observing that the problems admit a decomposition into a number d of subproblems, all of the same type, whose area-time complexity is determined by bisection flow and it is therefore described by a relation of the form $at^2 = \Omega(I^2)$, for $t \in [t_{min}, t_{max}]$.

For short keys, the $d = n/r$ subproblems are transcoding of multisets of size r, $I = \theta(r)$, $t_{min} = O(\log n)$, and $t_{max} = O(\sqrt{r})$. For long keys, the $d = k/\log n$ subproblems are ranking and permuting of n $\log n$-bit substrings of the keys, $I = \theta(n \log n)$, $t_{min} = O(\log n)$ and $t_{max} = O(\sqrt{n \log n})$.

If the network is equipped with d modules, each capable of processing a subproblem in area a and time t, the overall performance of the network is given by $A = da$, $T = t$ so that $AT^2 = \Omega(d\ I^2)$ for $t \in [t_{min}, t_{max}]$. This is the "fast" regime.

If the network is equipped with $d_2 < d$ modules, then the best strategy is to have d/d_2 subproblems processed by each module, and to realize the modules as small as possible, i.e. with $t = t_{max}$. In this situation $A = d_2 a$, $T = t_{max} d/d_2$, and $AT = da_{max} = \theta(d I^2 / t_{max})$. For $1 \leq d_2 \leq d$, $T \in [t_{max}, dt_{max}]$ and we obtain the "slow" regime.

square tessellation technique subsumes the bisection technique as a special case, and, for some problems like cyclic shift, sorting of short words, and sorting of long words (and probably many others), it provides much stronger bounds.

A very interesting finding has been that the form of the bounds obtained by the tessellation technique depends upon the computational mechanism forcing the information exchange.

One mechanism, extensively investigated by several authors in the two-processor environment, is present when some of the output variables of one processor are functions of the input variables of the other processor. If $\mathcal{U}$ is the set of I/O variables we wish to consider, we denote by $I(m)$ the minimum information exchanged by the two processors when m variables of $\mathcal{U}$ are assigned to one processor and $|\mathcal{U}|-m$ are assigned to the other. With this notation, the square tessellation bound can be stated as $AT^2 = \Omega(I^2(m)/m)$. For each problem there is a value of m that maximizes $I^2(m)/m$, and therefore gives the best lower bound. Thus, it becomes interesting to investigate the structure of I as a function of m, rather than restricting the attention to values of m that are constant fractions of $|\mathcal{U}|$, as it is done in most of the current literature.

Another mechanism, considered here for the first time, and referred to as saturation, occurs when one of the two processors fills up its memory locations during the computation, and sends some information to its mate for temporary storage. The analysis of the information exchanged by the cell of a tessellation because of saturation yields lower bounds on the AT measure. For specific problems such as cyclic shift, sorting of short words, and sorting of long words, both an AT^2 and an AT bound can be obtained. The latter dominates in slow computations while the former dominates in fast ones, so that the computation really exhibits two different regimes.

To see how well the relevant aspects of the computation are captured by our lower-bound techniques we have turned our attention to upper bounds.

We have begun with a careful study of a special case of sorting, where the key length is $k = \log n + \theta(\log n)$, hitherto the center of considerable attention in the VLSI literature. We have designed optimal sorters in the entire range of meaningful computation times, by developing the

CHAPTER 8

CONCLUSIONS

In this thesis we have studied the minimum area $A = \alpha_{n,k}(T)$ required to layout a circuit that sorts n k-bit numbers in time T. For the entire range of T, n and k we have derived lower bounds and proposed optimal or near-optimal designs.

In spite of the extensive investigations devoted to sorting in the past two decades, many new facets of this problem have been revealed by its analysis in the VLSI model of computation. As we have seen, the nature of sorting varies considerably with the relative size of the multiset being sorted and the size of the universe from which the keys are drawn. Indeed, at least three intervals of key lengths can be identified for each of which the area-time complexity is dominated by different phenomena, so that different algorithms, architectures, and lower- bound arguments are appropriate for each interval.

Not only our analysis has deepened our understanding of sorting, a problem by itself fundamental both from a theoretical and from a practical point of view, but it has also lead to the development of tools and methodologies of general interest for the field of VLSI computation theory.

Since the origins of VLSI complexity theory it has been clear that VLSI computations are dominated by the flow of information in the planar chip. Traditionally, area-time lower bounds have been based on the information I exchanged across a suitable bisection of the computation graph, and have the form $AT^2 = \Omega(I^2)$.

We have introduced the square tessellation technique, a powerful tool to establish lower bounds based on the information exchanged across the boundary of a suitable set of square cells that tessellate the layout region. An appropriate choice of the cell size enables us to capture the information flow at the level where the bandwidth necessary to sustain it poses the heaviest demand in terms of area. The

Comparing the results of Theorem 7.3 with lower bounds 7.57 and 7.58 we can make the following observations. For $T \in [\Omega(\log d \log n), O(\log d \sqrt{n \log n})]$ there is an $O(\log^2 d) = O(\log^2(k/\log n))$ gap between upper and lower bounds. For $T \in [\Omega(\log d \sqrt{n \log n}), O(d \sqrt{n \log n})]$ there is an $O(\log(d \sqrt{n \log n}/T))$ gap, which diminishes with T, and becomes $O(1)$ for the slowest design, as we have already noticed above. This design simultaneously attains the $A = \Omega(n \log n)$ lower bound on the layout area.

Although the small size of the gap indicates that steps in the right direction have been made in the analysis of the problem of sorting long keys, further work is needed to obtain a complete characterization of the area-time complexity. Besides the gap itself, another problem is that our fastest design achieves $T = O(\log n \log d)$, while the lower bound on computation time is $T = \Omega(\log n + \log d)$.

Two observations on the results of this section are worth mentioning.

(i) The sorter we have described has all I/O ports on the boundary, a condition that has not been exploited in deriving the lower bounds.

(ii) For $n = 2$, the sorter becomes a comparator-exchanger. Indeed, all the modules of the network degenerate to simple gates and the resulting circuit is very simple. Thus, considering that when $n = 2$, then $\log n = 1$, and $d = k$, Theorem 7.3 yields the following interesting corollary.

Theorem 7.4. A comparator-exchanger can be constructed with the following performance:

$$A = O(k/T \log(k/T)), \text{ for } T \in [\Omega(\log k), O(k)]. \qquad 7.78$$

Proof. Relation 7.78 is obtained from 7.77 after substitutions $d = k$, and $n = 2$. □

A comparison with the result of Theorem 4.22 shows that the comparator-exchanger of Theorem 7.4 is area-time optimal in the entire range of possible computation times.

$$A = O(k\ n^2/\log n) = O(d\ n^2) \tag{7.72}$$

and

$$T = O(\log n\ \log(k/\log n)) = O(\log n \log d). \tag{7.73}$$

If, instead, we want to minimize the area, we need to minimize both b and d_2. This is achieved when $b = O(\sqrt{n \log n})$, and $d_2 = 1$ so that each tree degenerates to a single node and the buffers disappear. The performance of this design is given by

$$A = O(n \log n), \tag{7.74}$$

and (since $d_1 = d/d_2 = d$)

$$T = O(d\ \sqrt{n \log n}). \tag{7.75}$$

This slow design is indeed optimal, as it achieves bound 7.58.

Between the fastest and the slowest design there is an interesting spectrum of intermediate behaviors, as stated by the following theorem.

Theorem 7.3 An (n,k)-sorter can be constructed, for $k \geq 2 \log n$, with the following performance:

$$AT^2 = O(d \log^2 d\ (n \log n)^2) \text{ for } T \in [\Omega(\log d\ \log n), O(\log d\ \sqrt{n \log n})], \tag{7.76}$$

and

$$AT = O(d \log(d\ \sqrt{n \log n}/T)(n \log n)^{3/2}) \text{ for } T \in [\Omega(\log d\ \sqrt{n \log n}), O(d\ \sqrt{n \log n})]. \tag{7.77}$$

Proof. The proof is essentially based on simple manipulations of Equations 7.68 and 7.69. More specifically, the AT^2 behavior is obtained by choosing for d_1 the value d_1' such that $d_1' = 2 \log(k/(\log n\ d_1'))$, according to Equation 7.70, and letting b vary in the range $[\Omega(\sqrt{n \log n}), O(n)]$, while d_1 is held fixed. The AT behavior is obtained by choosing $b = \Theta(\sqrt{n \log n})$, and letting d_1 vary in the interval $[d_1', d]$. □

7.4.3 Area-Time Performance

We now assume that all the modules of the network have an $O(b)\times O(b)$ area, and that all the connections are realized with $O(b)$ bandwidth. Thus, the entire network can be easily laid out in area

$$A = O(d_2 \log d_2 b^2) \tag{7.68}$$

As to the computation time, it is easy to see that the basic operations of the modules can be all performed in $O(n\log n/b)$ time, provided that $b \in [\Omega(\sqrt{n\log n}), O(n)]$. Indeed, the operations that are computationally harder are all of the sorting type, and are performed on sequences of n keys whose length is a small multiple of $\log n$.

Considering that the global number of basic steps is proportional to $2 \log d_2$, (the depth of the pipe), plus d_1, (the number of wavefronts) the global computation time is

$$T = O((d_1 + 2 \log d_2) n\log n / b). \tag{7.69}$$

It remains to select the values for d_1, d_2 and b to optimize the area time performance, recalling the constraints $k = d_1 d_2 \log n$ and $b \in [\Omega(\sqrt{n\log n}), O(n)]$.

We begin by considering some special cases in order to gain some intuition. Suppose, for instance, that we desire to obtain the fastest possible execution. It is clear that we have to maximize b and to minimize $d_1 + 2 \log d_2$. For b, the maximum value in the permissible range is $b = \theta(n)$. To minimize $d_1 + 2 \log d_2$, under the constraint $d_1 d_2 = k / \log n$, we should choose $d_1 = 2$. However, decreasing d_1 increases d_2 and therefore the area (see7.68). Moreover, when d_1 becomes smaller than $2 \log d_2$ there is no gain in the order of the computation time. Thus, we choose for d_1 the value that satisfies the equation

$$d_1 = 2 \log d_2 = 2 \log (k / (\log n\ d_1)) \tag{7.70}$$

or, more precisely, the integer part of it. Thus

$$d_2 = k / (d_1 \log n) = O((k / \log n) / \log (k / \log n)). \tag{7.71}$$

In conclusion, we obtain

slightly different from an internal-node module. More specifically, the basic operation of a leaf consists of receiving the keys in a block, say $X_0(h), \ldots, X_{n-1}(h)$, and of computing their ranks, $rank\ (X_0(h)), \ldots, rank\ (X_{n-1}(h))$.

The basic operation of an internal node consists in computing the ranks of the elements in the sequence obtained by pairwise concatenation of the sequences produced by the two offspring nodes.

A special role is played by the root module, which ranks the concatenation of three sequences, namely (from left to right): the ranks computed at the previous step, the ranks produced by the left son, and those produced by the right son.

(b) *The Broadcast-and-Sort Tree.* This component is also a fully balanced binary tree on d_2 leaves. The root of this tree is connected to the root of the ranking tree, from which it receives the sequence of the ranks. The internal nodes are simple modules (buffers) that duplicate the input sequence so that, after $\log_2 d_2$ levels, d_2 copies of the ranking sequence corresponding to a given input wavefront are available, one for each leaf.

The leaves are essentially sorting modules. Their basic operation consists in permuting the keys of a given block according to the corresponding ranks, as described in step 4 of the algorithm.

(c) *The Buffers.* If we consider a given block $X(h)$, we see that there is an interval of time elapsing between the time when the block is input by a leaf of the ranking tree, and the time when the same block is input by a leaf of the broadcast-and-sort tree, to be rearranged in the final order. During this interval the keys of the block are temporarily stored in buffers, connected as a linear array of $2 \log d_2 - 1$ modules.

Assuming that all the modules of the network perform their basic operation in the same time (which can always be achieved without affecting the asymptotic performance), the buffers guarantee that the sequence of keys of a given block reaches a leaf of the sorting tree at the same time of the corresponding sequence of ranks.

REFERENCES

[AA 80] H. Abelson and P. Andreae, "Information transfer and area-time trade-offs for VLSI multiplication," *Communications of the ACM*, vol. 23, n 1, pp. 20-22; January 1980.

[AG 83] A. Aggarwal, "On I/O placement in VLSI circuits," *Proc. 21st Annual Allerton Conference on Communication, Control, and Computing*, Monticello, IL, pp. 236-243; October 1983.

[AKS 83] M. Aitai, J. Komlos and E. Szemeredi, "An O(nlogn) sorting network," *Proc. 15th Annual ACM Symposium on Theory of Computing*, Boston, MA, pp. 1-9; April 1983.

[AUY 83] A.V.Aho, J.D. Ullman and M. Yannakakis, "On notions of information transfer in VLSI circuits," *Proc. 15th CAM Symposium on Theory of Computing*, Boston, MA, pp. 133-139; April 1983.

[Ba 68] K. E. Batcher, "Sorting networks and their applications," *Proc. AFIPS Spring Joint Computer Conference*,vol. 32, pp. 307-314; April 1968.

[Be 64] V. E. Benes, "Optimal rearrangeable multi-stage connecting networks," *Bell Syst. Tech. J.*, vol. 43, n. 4, pp. 1641-1656; July 1964.

[BG 82] M. P. Brent and L. M. Goldschlager, "Some area-time tradeoffs for VLSI," *SIAM J. on Comput.*, vol. 11, n. 4, pp. 737-747; November 1982.

[BJ 84] G. Bilardi and X. Jin, "Permutation exchange graphs that emulate the binary cube," *Mathematical System Theory*, vol. 17, n. 3, pp. 193-198; June 1984.

[BK 80] R. P. Brent and H. T. Kung, "On the area of binary tree layouts," *Information Processing Letters*,vol. 11, n. 1, pp. 46-48; August 1980.

[BK 81] R. P. Brent and H. T. Kung, "The chip complexity of binary arithmetic," *Journal of the ACM*, vol. 28, n. 3, pp. 521-534; July 1981.

[BK 82] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Trans. on Comp.*, vol. C-31, n. 3, pp. 260-264; March 1982.

[BL 84] S. N. Bhatt and F. T. Leighton, "A framework for solving VLSI graph layout problems," *J. of Comp. and Syst. Sci.*, vol. 28, n. 2, pp. 300-342; April 1984.

[BP 84a] G. Bilardi and F. P. Preparata, "An architecture for bitonic sorting with optimal VLSI performance," *IEEE Trans. Comp.*, vol. C-33, n. 7, pp. 646-651; July 1984.

[BP 84b] G. Bilardi and F. P. Preparata, "A minimum area VLSI network for O(logN) time sorting," *Proc. 16th Annual ACM Symposium on Theory of Computing*, Washington, D. C., pp. 64-70; April 1984.

[BP 84c] G. Bilardi and F. P. Preparata, "The VLSI optimality of the AKS sorting network," *Information Processing Letters*, to appear.

[BPP 82] G. Bilardi, M. Pracchi, and F. P. Preparata, "A critique of network speed in VLSI models of computation," *IEEE J. of Solid-State Circuits*, vol. SC-17, n. 4, pp. 696-702; August 1982.

[ES 84] G. Bilardi and M. Sarrafzadeh, "Optimal discrete Fourier transform in VLSI," *International Workshop on Parallel Computing and VLSI*, Amalfi, Italy, May 1984.

[CM 81] B. Chazelle and L. Monier, "A model of computation for VLSI with related complexity results," *Proc. 13th Annual ACM Symposium on Theory of Computing*, Milwaukee, WI, pp. 318-325; May 1981.

[DGS 84] P. Duris, Z. Galil and G. Schnitger, "Lower bounds on communication complexity," *Proc. 16th Annual ACM Symposium on Theory of Computing*, Washington, D. C., pp. 81-91; August 1984.

[GHT 79] L. J. Guibas, H. T. Kung and C. D. Thompson, "Direct VLSI implementation of combinatorial algorithms," *Proc. Conference on VLSI Architecture, Design, Fabrication*, Calif. Inst. of Techn.; January 1979.

[Jh 80] R. B. Johnson, "The complexity of a VLSI adder," *Information Processing Letters*, vol. 11, n. 2, pp. 92-93; October 1980.

[JK 84] J. Ja' Ja' and V. K. P. Kumar, "Information transfer in distributed computing with applications to VLSI," *Journal of the ACM*, vol. 31, n. 1, pp. 150-162; January 1984.

[KL 78] H. T. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," *Symposium on Sparse Matrix Computations*, Knoxville, TN, pp. 256-282; November 1978.

[KLLM 83] D. Kleitman, F. T. Leighton, M. Lepley, and G. L. Miller, "An asymptotically optimal layout for the shuffle-exchange graph," *J. of Comp. and Syst. Sci.*, vol. 6, n. 3, pp. 339-361; June 1983.

[Km 83] V. K. P. Kumar, *Communication Complexity of Various VLSI Models*, Ph.D. Thesis, Dept. of Comp. Science, Pennsylvania State University; August 1983.

[KR 82] K. Keutzer and E. Robertson, "The M-shuffle as an interconnection network for SIMD machines," *Proc. of 20th Annual Allerton Conference on Communication, Control, and Computing*, Monticello, IL, pp. 264-271; October 1982.

[Ku 82] H. T. Kung, "Why systolic architectures?" *Computer Magazine*, vol. 15, n. 1, pp. 37-46; January 1982.

[L 81a] F. T. Leighton, *Layouts for the shuffle-exchange graph and lower bound techniques*, Ph.D. Thesis, Dept. of Mathematics, MIT, August 1981.

[L 81b] F. T. Leighton, "New lower bound techniques for VLSI," *Proc. 22nd Annual Symposium on the Foundations of Computer Science*, Nashville, TN, pp. 1-12; October 1981.

[L 82] F. T. Leighton, "A layout strategy which is provably good," *Proc. 14th Annual ACM Symposium on Theory of Computing*, San Francisco, CA, pp. 85-98; May 1982.

[L 83] F. T. Leighton, "Parallel computation using meshes of trees," submitted for publication.

[L 84] F. T. Leighton, "Tight bounds on the complexity of parallel sorting," *Proc. 16th Annual ACM Symposium on Theory of Computing*, Washington, D. C., pp. 71-80; April 1984.

[Lo 83] M. C. Loui, "The complexity of sorting on distributed systems," *Tech. Report ACT-39*, Coordinated Science Laboratory, University of Illinois,Urbana, IL; September 1983.

[Ls 80a] C. E. Leiserson, *Area efficient VLSI computation*, Ph.D. Thesis, Dept. of Comp. Science, Carnegie-Mellon University; November 1980.

[Ls 80b] C. E. Leiserson, "Area-efficient graph layouts (for VLSI)," *Proc. 21st Annual Symposium on Foundations of Computer Science*, Syracuse, NY, pp. 270-281; October 1980.

[LS 81] R. J. Lipton and R. Sedgewick, "Lower bounds for VLSI," *Proc. 13th Annual ACM Symposium on Theory of Computing*, Milwaukee, WI, pp. 300-306; May 1981.

[Me 83] K. Mehlhorn, "AT^2 optimal VLSI integer division and integer square rooting," submitted for publication.

[MC 79] C. A. Mead and L. Conway, *Introduction to VLSI Systems*, Reading, MA, Addison-Wesley; July 1979.

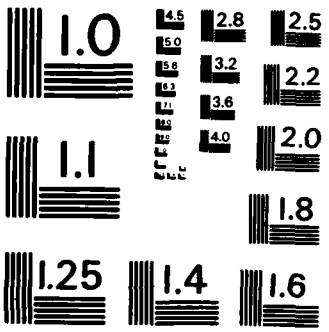
[MP 75] D. E. Muller and F. P. Preparata, "Bounds to complexities of networks for sorting and switching," *Journal of ACM*, vol. 22, n. 2, pp. 195-201; April 1975.

[MP 83] K. Mehlhorn and F. P. Preparata, "Area-time optimal VLSI integer multiplier with minimum computation time," *Information and Control*, vol. 58, nos. 1-3, pp. 137-156; July 1983.

[MS 82] K. Mehlhorn and E. M. Schmidt, "Las Vegas is better than determinism in VLSI and distributed computing," *Proc. 14th Annual ACM Symposium on Theory of Computing*, San Francisco, CA, pp. 330-337; May 1982.

AD-A161 562 THE AREA-TIME COMPLEXITY OF SORTING(U) ILLINOIS UNIV AT 3/3
URBANA APPLIED COMPUTATION THEORY GROUP G BILARDI
DEC 84 ACT-52 N00014-84-C-0149

UNCLASSIFIED F/G 12/1 NL



[Mu 82] S. Muroga, *VLSI System Design*, J. Wiley, New York; 1982.

[NMB 83] D. D. Nath, S. N. Maheshwari and P. C. P. Bhatt, "Efficient VLSI networks for parallel processing based on orthogonal trees," *IEEE Trans. on Comp.*, vol. C-32, n. 6, pp. 569-581; June 1983.

[NS 79] D. Nassimi and S. Sahni, "Bitonic sort on a mesh-connected parallel computer," *IEEE Trans. on Comp.*, vol. C-28, n. 1, pp. 2-7; January 1979.

[NS 82] D. Nassimi and S. Sahni, "Parallel permutation and sorting algorithms and a new generalized connection network," *Journal of ACM*, vol. 20, n. 3, pp. 642-667; July 1982.

[P 78] F. P. Preparata, "New parallel sorting schemes," *IEEE Trans. Comp.*, vol. C-27, n. 7, pp. 669-673; July 1978.

[P 84] F.P. Preparata, "VLSI algorithms and architectures," *Proceedings of 11th Symposium on Mathematical Foundations of Computer Science*, Praha, Czechoslovakia; September 1984.

[Pe 77] M. C. Pease, "The indirect binary n-cube microprocessor array," *IEEE Trans. on Comp.*, vol. C-26, n. 5, pp. 458-473; May 1977.

[PS 84] C. H. Papadimitriou and M. Sipser, "Communication complexity," *J. Compu. System Sci.*, vol. 28, n. 2, pp. 260-263; April 1984.

[PV 80] F. P. Preparata and J. Vuillemin, "Area-time optimal VLSI networks for multiplying matrices," *Information Processing Letters*, vol. 11, n. 2, pp. 77-80; October 1980.

[PV 81a] F. P. Preparata and J. Vuillemin, "The cube-connected-cycles: A versatile network for parallel computation," *Communications of the ACM*, vol. 24, n. 5, pp. 300-309; May 1981.

[PV81b] F. P. Preparata and J. Vuillemin, "Area-time optimal VLSI networks for computing integer multiplication and discrete Fourier transform," *Proc. of I. C. A. L. P.*, Haifa, Israel, pp. 29-40; July 1981.

[Sa 79] J. E. Savage, "Area-time tradeoffs for matrix multiplication and related problems in VLSI models," *Proc. of the 17th Annual Allerton Conference on Communications, Control, and Computing*, Monticello, IL, pp. 670-676; October 1979.

[Se 79] C. L. Seitz, "System timing," in *Introduction to VLSI Systems*, C. Mead and L. Conway, Eds., Reading, MA, Addison-Wesley, ch. 7; July 1979.

[Sg 84a] A. Siegel, "Optimal area VLSI circuits for sorting," submitted for publication.

[Sg 84b] A. Siegel, "Tight area bounds and provably good AT^2 bounds for sorting circuits," *Tech. Report #122* Courant Institute, New York University; June 1984.

[St 71] H. S.Stone, "Parallel processing with the perfect shuffle," *IEEE Trans. on Comp.*, vol. C-20, n. 2, pp. 153-161; February 1971.

[T 80] C. D. Thompson, *A compiexity theory for VLSI*, Ph.D. Thesis, Dept. of Comp. Science, Carnegie-Mellon University; August 1980.

[T 83a] C. D. Thompson, "Fourier transforms in VLSI," *IEEE Trans. Comp.*, vol. C-3, n. 11, pp. 1047-1057; November 1983.

[T 83b] C. D. Thompson, "The VLSI complexity of sorting," *IEEE Trans. Comp.*, vol. C-32, n. 12, pp. 1171-1184; December 1983.

[TK 77] C. D. Thompson and H. T. Kung, "Sorting on a mesh-connected computer," *Communications of the ACM*, vol. 20, n. 4, pp. 263-271; April 1977.

[U 83] J. D. Ullman, *Computational Aspects of VLSI*, Computer Science Press; 1983.

[Va 81] L. G. Valiant, "Universality considerations in VLSI circuits," *IEEE Trans. on Comp.*, vol. C-30, n. 2, pp. 135-140; February 1981.

[Vu 83] J. Vuillemin, "A combinatorial limit to the computing power of VLSI circuits," *IEEE Trans. on Comp.*, vol. C-32, n. 3, pp. 294-300; March 1983.

[Y 79] A. C. C. Yao, "Some complexity questions related to distributive computing," *Proc. 11th Annual ACM Symposium on Theory of Computing*, Atlanta, GA, pp. 209-213; April 1979.

[Y 81] A. C. C. Yao, "The entropic limitations on VLSI computations," *Proc. 13th Annual ACM Symposium on Theory of Computing*, Milwaukee, WI, pp. 308-311; April 1981.

VITA

Gianfranco Bilardi was born in Reggio Calabria, Italy, on March 8, 1956. He received the Laurea in Ingegneria Elettronica degree from Università di Padova in 1978 and the M.S. degree in Electrical Engineering from the University of Illinois at Urbana-Champaign in 1982, where he was a Research Assistant in the Coordinated Science Laboratory until 1984. He was awarded an International Rotary Fellowship in 1980, and an IBM Graduate Fellowhip in 1982 and 1983.

END

FILMED

1-86

DTIC